March 2013

Multi-Point Stride Coverage: A New Genre of Test Coverage Criteria

Mohammad Mahdi Hassan
The University of Western Ontario

Supervisor
Associate Professor Dr. James H. Andrews
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

MULTI-POINT STRIDE COVERAGE: A NEW GENRE OF TEST COVERAGE CRITERIA

(Spine title: Multi-point Stride Coverage)
(Thesis format: Monograph)

by

Mohammad Hassan

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

We introduce a family of coverage criteria, called Multi-Point Stride Coverage (MPSC). MPSC generalizes branch coverage to coverage of tuples of branches taken from the execution sequence of a program. We investigate its potential as a replacement for dataflow coverage, such as def-use coverage. We find that programs can be instrumented for MPSC easily, that the instrumentation usually incurs less overhead than that for def-use coverage, and that MPSC is comparable in usefulness to def-use in predicting test suite effectiveness. We also find that the space required to collect MPSC can be predicted from the number of branches in the program.

Keywords: Software Testing; Control Flow Coverage; Data Flow Coverage; Partial Execution Pattern

Co-Authorship Statement

There was an existing branch code instrumentor developed by my supervisor Jamie Andrews in JAVA; later he modified it to support the bitset and hashset implementations of MPSC. I added some modules to support switch structure and fixed some bugs. I developed the scripts to collect data and to run mutation testing automatically. We used some existing tools like DuaF, to collect def-use coverage, by Raúl Santelices, and Mutgen, a mutation generator, by Jamie Andrews.

For data analysis I wrote several programs using R according to my supervisor guidance. He wrote some sections on data analysis for our conference submission (accepted in ICSE 2013) which I adapted in this thesis manuscript.

My supervisor provides the basic definitions related to MPSC described in Chapter 2. We included multiple points instead of two points as I proposed.

Acknowlegements

Thank you to Gregg Rothermel and the University of Nebraska-Lincoln for the SIR repository, the source of our subject programs. Many thanks to Raúl Santelices for sharing and discussing his Java DU-coverage tools, and to Michael Ernst for his comments on an earlier draft. This research is supported by NSERC and by an Ontario government OGS scholarship to Mohammad Mahdi Hassan.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Like any product, software needs to be tested to ensure quality. There are different kinds of testing emphasizing different aspects of software. For instance, functional testing targets functionality of a program, regression testing ensures that a change in an existing system did not introduce new faults, and robustness testing checks the software's robustness when the load is high.

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. Every test case incurs some cost like design cost, configuration cost, execution cost and analysis cost. Beside finding faults, one of the target in software testing is to create a optimal test suite that test the software sufficiently and thoroughly.

The purpose of *structural* testing is to check some aspects (statements, conditions etc.) of the source code that constitutes the software. The idea behind it is to ensure that the software under test is checked thoroughly. As an example, if a statement in the source code is not executed by a test suite, then the test suite provides no evidence of its correctness or incorrectness, as it is difficult to measure its impact on the final output without executing that line of code; this is true even for a moderate-sized program. The strategy in structural testing is to cover a high percentage of the targeted features of the source code and use this measure to indicate testing thoroughness. This measure can be used as a stopping criterion in test case generation process. In popular terms it is also known as code coverage testing. Over many studies in the last several decades it has been shown that using code coverage as a catalyst in testing increases the chance of finding software faults in the testing phase; it also helps localize them.

Structural coverage measures measure how much of a program's structure a test suite exercises. Some forms of coverage criteria have been used in software testing for decades to judge whether test suites are thorough enough [65]. In larger software companies (e.g. IBM and Microsoft) it has become common to show coverage related metrics information while reporting unit testing [66].

In broad terms, structural coverage criteria can be divided into two different domains: "Control Flow Coverage" (CFC) and "Data Flow Coverage" (DFC) [54]. CFC includes measures like branch coverage, which requires that each direction of each branch be taken by some test case. DFC includes measures like "def-use coverage", which requires that every def-use path (path from an assignment of a value to a variable to a use of that assigned value) is taken by some test case.

1.1 Motivation

In general it is accepted that DFC is superior to CFC in terms of finding and locating faults [38]. There is a tradeoff between them, as DFC is complicated to implement and its measurement incurs more overhead than CFC, so in practice some form of CFC is accepted and widely used in the software industry. In our research, we bridge the gap between the measures by introducing a family of CFC criteria which we call "Multi-Point Stride Coverage" (MPSC). Informally, instrumentation for MPSC with gap g and p points records the coverage of tuples $(b_1, \ldots, b_p)$ of branches taken, where each branch in the tuple is the one taken g branches after the previous one[1]. We believe that MPSC could be less costly than DFC but could be as

[1] A detail description of MPSC is given in chapter 2

effective as DFC.

Collecting coverage data is a key aspect of structural testing; it is used to generate relevant statistics, e.g. percentage of coverage by individual test cases or test suites. Coverage data can be collected with or without preserving the sequence as a trace. For our experiments, we have developed a tool that can collect branch coverage in three different languages (C, C++ and Java). It also preserves execution sequence which is a key aspect to our proposed coverage criterion. This is important if we want to understand the pattern of execution that may represent the logic behind the code in more detail than traditional coverage criteria. Heimdahl et al. [48] suggests test cases that technically provide the right coverage for traditional code coverage criteria, still may not exercise the logic of the software. One of the important aspects of DFC criteria is that they inherently capture aspects of the execution pattern; with MPSC, we devise a mechanism to capture different forms of patterns by adjusting parameters g (i.e. length of the gap between two execution points) and p (i.e. number of execution points considered).

The main difference between CFC and DFC is the use of data containers (i.e. variables); to check any DFC criterion there is a need for a backward referencing mechanism (i.e. to know where the variables are defined or last modified). In CFC this is not an issue, as those defined criteria (statements, branches etc) can be collected by just following the flow of execution. The situation becomes more complex for DFC if the data containers are indirect (i.e. pointers) or have a complex structure (e.g. a C, C++ or Java class or structure). So DFC implementation is complex, and the processing overhead is very high compared to CFC. On the other hand, DFC inherently covers two highly related arbitrarily distant points in an execution path, which makes it a stronger candidate to reveal implementation flaws and bugs of a program. In MPSC we try to capture the good part of both domains. MPSC captures multiple points in an execution path according to their execution sequence, but at the same time it reduces the complexity of backward referencing. Like CFC, just following an execution flow with a length N buffer (i.e. to store the last N executions) we can capture MPSC of different varieties.

1.2 Research Questions

We believe that MPSC could be a coverage criterion which is as effective as def-use coverage, but easier to instrument and more efficient to collect. To investigate whether this is the case, we designed and performed experiments. In this section we give a brief description of four research questions that are necessary to understand MPSC and its effectiveness. Our research addresses four main research questions:

- **RQ1.** How many MPSC tuples typically need to be collected for a program, and how is that number related to other program metrics?
- **RQ2.** What is the relationship between MPSC and DFC criteria such as def-use?
- **RQ3.** Does MPSC lead to a more accurate assessment of test suite effectiveness than def-use?
- **RQ4.** What is the performance overhead of collecting MPSC, compared to the uninstrumented program and to DFC?

1.3 Key Findings

We addressed these questions by experimentation on 15 programs of various sizes in three commonly-used programming languages (C, C++ and Java). Our main findings on these programs are summarized below.

- Every program is associated with a constant *C* which allows us to accurately predict how much storage to allocate for MPSC data collection, given the number of branches in the program.
- The def-use coverage of test suites is strongly correlated with their branch coverage, raising questions about how distinct def-use coverage is from branch coverage in practical settings.

- MPSC allows us to predict the effectiveness of a test suite with a similar or higher level of accuracy than def-use.
- The instrumentation for MPSC is usually more efficient than that for def-use coverage.
- The simplest member of the MPSC family of criteria is often the most useful member. However, the family as a whole provides the user with a wide choice of useful coverage criteria.

1.4 Thesis Structure

We organize the chapters as follows. In Chapter 2, we give some basic definitions. In Chapter 3, we discuss related work. In Chapter 4, we describe the design and results of our empirical study of the basic properties of MPSC (RQ1). In Chapter 5, we describe the design and results of experiments to determine the relationships between MPSC and def-use coverage (RQ2 and RQ3). In Chapter 6, we describe the design and results of experiments we did to measure the performance overhead of MPSC compared to def-use (RQ4). In Chapter 7, we present an overall discussion of the results. In Chapter 8, we conclude and suggest future work.

Chapter 2

Definitions

In order to facilitate later discussion, we will revisit some traditional coverage criteria which we consider in our research; then we will discuss briefly the benefit of considering sequence while collecting a program's structural features. Finally, we will give a formal definition of MPSC (Multi Point Stride Coverage), followed by more related definitions.

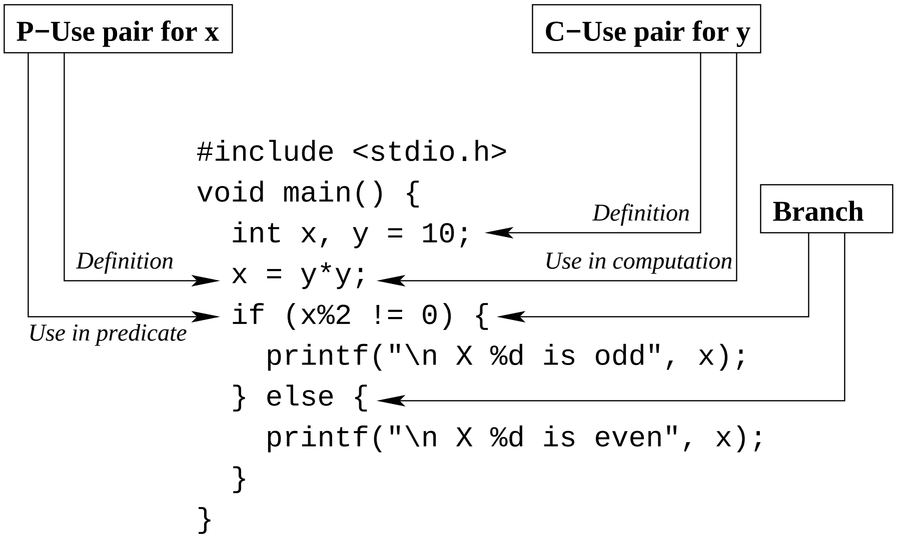
2.1 Branch, Def-use and its variations

A *branch* in a program is one direction of an `if`, `for` or `while` decision, or one `case` of a `switch` statement[1]. The *branch coverage* of a test suite (a set of test cases) is the proportion of branches that are executed by at least one test case [65].

A statement is a *definition* or *def* of a variable x if it assigns x a new value. A statement is a *use* of a variable x if it contains a reference to the current value of x. Two statements s_1 and s_2 constitute a *def-use pair* for x if s_1 is a def of x, s_2 is a use of x, and there is a path π from s_1 to s_2 that does not pass through any other defs of x [79]. The *def-use coverage* of a test suite is the proportion of def-use pair paths π that are executed by at least one test case.

There are also two common forms of data flow coverage known as "C-use" (computation use) and "P-use" (predicate use). In C-use, a pair is created by a definition of a variable and

[1] We restrict our attention to C, C++ and Java. The concepts can be easily extended to other languages.



```
#include <stdio.h>
void main() {
  int x, y = 10;
  x = y*y;
  if (x%2 != 0) {
    printf("\n X %d is odd", x);
  } else {
    printf("\n X %d is even", x);
  }
}
```

Figure 2.1: A sample C code with some examples of different kinds of code coverages

its use in computational statements (for example, y= x * 2) whereas *P*-use considers the use in predicates (for example, if (x == 2). . .). We do not discuss in detail here issues such as the treatment of function parameters and the distinction between P-use and C-use, nor do we discuss the problem of infeasibility [26] in detail; the reader is referred to Rapps and Weyuker [79] and Horgan and London [49] for more detail. In Figure 2.1, we show these coverage criteria in a simple C program.

2.2 Considering Sequence while collecting program features

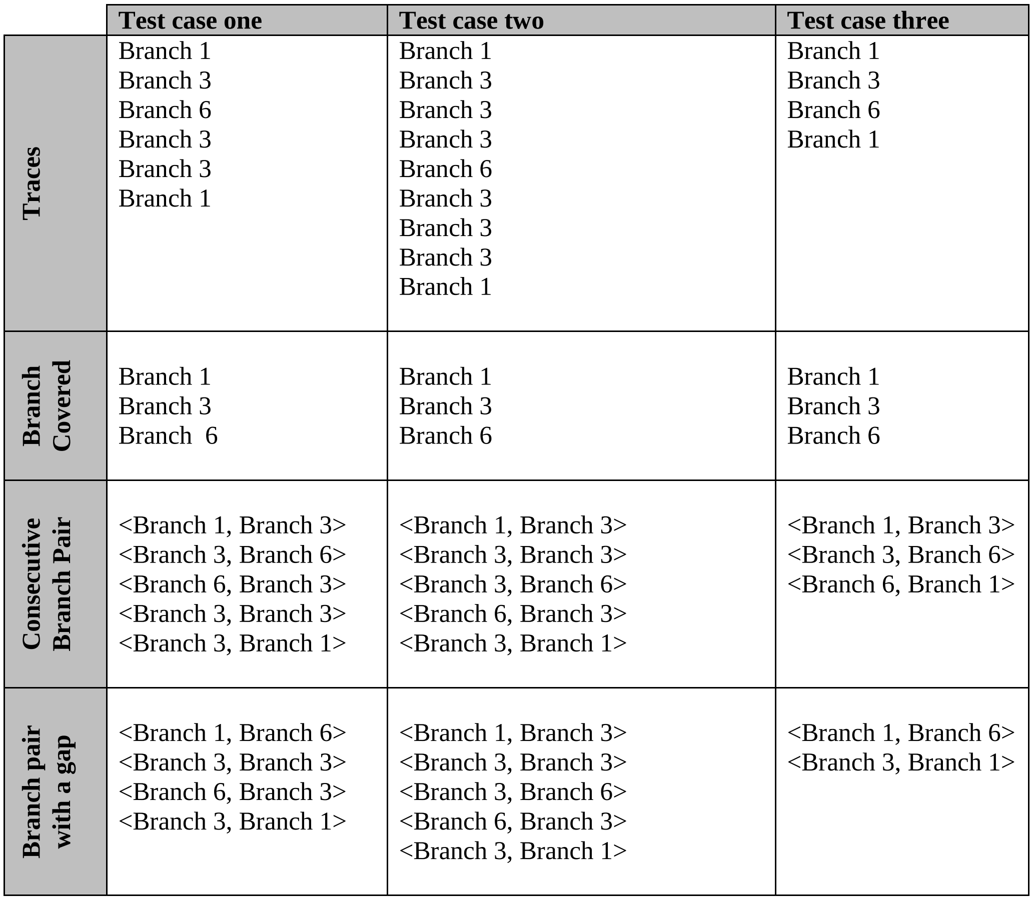
Researchers and software professionals collect features that include control flow, variable values, addresses, and dependencies to create a program profile [9]. These collected features are generally known as program-execution traces or logs. Using these traces, it is possible to identify program characteristics that are useful for improving aspects of program behavior including performance and correctness.

Feature collection and analysis can be performed intraprocedural or interprocedural. In intraprocedural analysis, features are considered within a specific procedure, whereas in interprocedural analysis the entire program is considered. Features such as data-flow coverage can be analyzed intraprocedurally by covering over all paths only; interprocedural data-flow coverage has to meet all valid paths [80]. Due to its scope, interprocedural analysis is more precise but much more complex than intraprocedural analysis. In our research, we define and collect our proposed feature (MPSC) interprocedurally.

A systematic collection of features can show the dynamic behavior of the program in detail, but it has a price to pay in performance overhead and memory allocation [18]. One of the ways to solve the resource-constraint problem is by reducing the output information; one example would be to report the covered branches only instead of the whole sequence of branches traversed in a program execution process. However, this approach reduces the scope and capability of the execution traces to their minimum, thus making this approach unsuitable for deeper analysis. A balance may be struck by preserving the sequence partially as a form of partial-execution pattern while reporting any structural features such as a branch or a statement. Thus, the length of the sequence can be manipulated in accord with needs and resource availability.

Furthermore, let us assume an arbitrary tool is able to record branch coverage information of a program where each branch is identified with a unique ID[2]. In Row One of Figure 2.2 we show examples of three traces for three different test cases. Each line reports the branch traversed, and the consecutive lines suggest the sequence of execution. Now if we just consider the branch coverage information, these three test cases look similar (as in Row Two, contain same set of branches), but if we slightly change the feature-collection mechanism to consider two consecutive branches as a pair, then the first two look similar but the third one looks different (as in Row Three). Finally, if we go a little bit further by considering a pair of branches with a gap of one branch between them, then all three test cases look much different (as in Row Three, contain different set of tuples). Here, the last two rows represent partial-

[2] In our research we develop a tool named as JQXZ where we identify each branch of a program by a unique ID consisting of four parts: a unique file number; a line number; the position of the first character of the branch; and a True or False direction which will be ignored for switch-case structure as there is only one direction in that structure.



	Test case one	Test case two	Test case three
Traces	Branch 1 Branch 3 Branch 6 Branch 3 Branch 3 Branch 1	Branch 1 Branch 3 Branch 3 Branch 3 Branch 6 Branch 3 Branch 3 Branch 3 Branch 1	Branch 1 Branch 3 Branch 6 Branch 1
Branch Covered	Branch 1 Branch 3 Branch 6	Branch 1 Branch 3 Branch 6	Branch 1 Branch 3 Branch 6
Consecutive Branch Pair	<Branch 1, Branch 3> <Branch 3, Branch 6> <Branch 6, Branch 3> <Branch 3, Branch 3> <Branch 3, Branch 1>	<Branch 1, Branch 3> <Branch 3, Branch 3> <Branch 3, Branch 6> <Branch 6, Branch 3> <Branch 3, Branch 1>	<Branch 1, Branch 3> <Branch 3, Branch 6> <Branch 6, Branch 1>
Branch pair with a gap	<Branch 1, Branch 6> <Branch 3, Branch 3> <Branch 6, Branch 3> <Branch 3, Branch 1>	<Branch 1, Branch 3> <Branch 3, Branch 3> <Branch 3, Branch 6> <Branch 6, Branch 3> <Branch 3, Branch 1>	<Branch 1, Branch 6> <Branch 3, Branch 1>

Figure 2.2: Capturing variation of test cases through pairing, an example of structural coverage manipulation

execution patterns; they not only cover the static structural features such as branches but also reveal execution patterns.

There is always a logic behind a program code. When a programmer writes a program he intentionally puts statements or decisions (i.e. codes) in a sequence that aligns with that logic. Therefore, sequence plays a crucial role in structuring a program. However, ignoring the sequence while collecting and reporting traditional structural coverage information can misrepresent both the variation and the thoroughness of the testing process. On the other hand, collecting all possible sequences (i.e. full path coverage) is infeasible even for a small program. There is a well-known strategy for faster performance commonly known as "Locality of Reference" [30] which suggests that in most cases program code refers to other code near its location. One of the applications of this approach is in the microprocessor design related to cache usage [41] or virtual memory [29]. This approach suggests that a program code is not simply a monolithic structure; rather, each chunk has its own concentrated meaning and purpose. Therefore, when it may not be feasible to record all possible path coverages, partial sequences may be used to represent the program instead. Our proposed coverage criteria allow one to partially observe the execution pattern of software. This may reveal the variation and thoroughness of the structural testing in a more succinct way.

2.3 Multi Point Stride Coverage(MPSC) and related definitions

We propose a family of coverage criteria which is a generalization of branch coverage. If b_1 and b_2 are branches, we say that a test case executes *gap-g branch stride* (b_1, b_2) if it executes b_1, and then when g branches later (interprocedurally) it executes b_2. We define the *gap-g branch stride coverage* of a test suite as the proportion of gap-g branch strides that are executed by at least one test case.

We generalize this to *multi-point stride coverage* (MPSC) by extending the pair (b_1, b_2) to a sequence of p points $(b_1, b_2, \ldots, b_p)$, each separated from the next by g branches (interprocedurally). We refer to the sequence of p points as an *MPSC tuple*. In order to handle the beginning and end of the execution sequence, we assume a sequence of *begin* pseudo-branches at the beginning of the sequence, and *end* pseudo-branches at the end. For instance, if a small program executes only the four branches $b1, b2, b3$, and $b4$ in that order, then there are six MPSC tuples with gap 1 and 3 points executed by the program: $(begin, begin, b_1)$, $(begin, b_1, b_2)$, (b_1, b_2, b_3), (b_2, b_3, b_4), (b_3, b_4, end), and (b_4, end, end).

MPSC is thus a family of coverage criteria, one for each value of gap size g and number of points p in a tuple. If $p = 1$, then MPSC is the same as branch coverage; if $g = 0$, then MPSC is equivalent to branch coverage, since all the points in an MPSC tuple with $g = 0$ are the same. MPSC can be extended to other kinds of coverage features (e.g., statements or conditions); the key consideration is that those features must be collected according to their execution sequence. In this thesis, we concentrate on MPSC based on branch coverage, since branch coverage is a moderately strong and commonly-used coverage measure.

We use $covset(S, t, g, p)$ to mean the set of unique MPSC tuples covered by test case t for program S when using gap g and p points. We define $ncov(S, t, g, p) = |covset(S, t, g, p)|$. We extend this to test suites, i.e. sets T of test cases, by defining $covset(S, T, g, p)$ as the union of all the sets $covset(S, t, g, p)$ for all $t \in T$, and $ncov(S, T, g, p) = |covset(S, T, g, p)|$.

We use $maxcov(S, g, p)$ to mean the maximum value that $ncov(S, T, g, p)$ can be for any test suite T. The precise value of $maxcov(S, g, p)$ cannot be determined in general, since whether a given branch in program S can be taken is undecidable. An obvious upper bound is N^p, where N is the number of branches in the program. Static analysis (analysis of source, executable or intermediate code) could provide a tighter upper bound, and dynamic analysis (execution of test cases) can provide a lower bound. In this thesis, we approximate $maxcov(S, g, p)$ by the size of the union of all the sets $covset(S, t, g, p)$ for all test cases t in a test pool for the program S.

We use $pcov(S, t, g, p)$ to mean $100 * ncov(S, t, g, p)/maxcov(S, g, p)$. This is the percentage of feasible tuples covered by the test case.

We use *averagencov* to mean $\sum_{t \in T} ncov(S, t, g, p)/|T|$. Here $|T|$ suggest total number of test cases in a suite. This is the average number of tuples per test case.

We use $maxcovratio(S, g, p)$ to mean the ratio $maxcov(S, g, p)/maxcov(S, 0, 2)$. We can think of this as the memory multiplication factor needed to collect MPSC coverage at gap g and p points, which is one of the cost factors for MPSC measurement.

A program can be instrumented for MPSC coverage more easily than for DU coverage since the coverage points are simply decisions. The natural data structure for the coverage data is a hash table; the optimal number of entries for the hash table can be determined from *maxcovratio*.

Chapter 3

Background and Related Work

Software testing helps develop higher quality software [16] through some specific and standard practices. In recent times, development cost has been going down due to the use of such mechanisms as software-product-line methods while testing cost has increased [35]. Testing effort takes 50 to 75 percent of total development cost [43], which indicates an enormous challenge and expense. Proper testing can save significant effort, increase product quality and reduce maintenance costs so as to ultimately increase customer satisfaction [2].

The idea of achieving software correctness [40, 67] through testing may look attractive, but a more realistic goal is to have a reliable testing strategy [51]. Testing can effectively find bugs but doesn't ensure their absence [31].

As early as 1963, Miller and Maloney suggested that the origins of mistakes in software are those portions of the program which were not tested properly [63]. The question is how to define a portion of the program. In a low-level sense, we can say a program is a collection of statements; on the other hand, we can consider the logical sequence behind those statements that binds them and makes them cohesive, ultimately giving them the capability to do meaningful tasks.

According to Ntafos [67], testing strategies can be classified into three different categories:

- **Structural Strategy:** This is also known as white-box testing [72]. One of the early models that has been used to understand the program is the flow graph, which is the basis of structural testing [79]. In general, the goal of structural testing is to cover the control structure of a program to some degree of thoroughness.

- **Black-box Strategy:** Here the program (or its component) is considered as a black box. Inputs and correctness of the corresponding outputs are the main concern. Knowledge related to the internal structure of a program is not essential; rather understanding of the purpose of the software (or its component) is necessary [74]. Examples of black-box testing include System [99], Functional [52] and Acceptance [53] testing.

- **Error-driven Strategy:** Based on known errors, this approach generates test cases which can capture them; mutation testing [28, 58] is a common example of this approach.

Our proposed coverage criteria (MPSC) falls into the category of structural testing (or white-box testing); we also use mutation as a validation method in our research. Detailed discussion is in section 3.8.

3.1 Code Coverage Measures

As open-ended testing is too vague to be viable, a major concern of software testing is to establish the thoroughness criterion of a test suite [40]. Structural code coverage measures, such as branch coverage, have long been studied as a means for evaluating the thoroughness of a test suite [65]. Tools that automatically evaluate given code coverage measures on a test suite are now commonly used in the industry [17, 105]. They are now increasing in importance, since they are used not only in test case construction by humans, but also in automatic test input generation [73, 8, 22, 25], test suite minimization and prioritization [81, 89, 32], and fault localization [59, 3].

Many coverage measures have been proposed; Zhu et al. [109] conducted a comprehensive survey on this topic. According to them, coverage measures can be grouped into the two broad classes of *controlflow-based* and *dataflow-based* measures. The former focuses on the flow of control from one statement to another, such as the "branch coverage" mentioned above. Dataflow-based measures, in contrast, focus on the flow of data from "definition" ("def") statements, which assign a value to a variable, to statements where that value of the variable is used. For instance, the statement `x = y` is a def for variable `x`. If that statement is on line 15 of a program, and line 25 of the program is "`if (x > 100)`", and it is possible for control to flow from line 15 to line 25 without executing any other defs of `x`, then the statements on line 15 and 25 form a "def-use pair", the dataflow equivalent of a branch of branch coverage. If line 15 and line 25 are in different conditional blocks, then 100% def-use coverage would require that the def-free path be executed, while 100% branch coverage might not require this.

We say that coverage measure A *subsumes* coverage measure B if every test suite that achieves A must necessarily achieve B. Frankl and Weyuker [37] proved subsumption and similar relations among a wide variety of coverage measures, including the fact that def-use coverage subsumes branch coverage. Ammann and Offutt extended this to many other coverage measures [5]. Ball showed that Predicate-Complete Test Coverage subsumes statement, branch, multiple condition and predicate coverage [10]. Note that for some particular cases a test suite which satisfies less rigorous coverage may still perform better (in terms of finding faults) than some suites which may satisfy more rigorous coverage[44]; Weyuker called them "misleading test[s]"[98]. Overall, subsuming coverage criteria guarantee better fault detection capability [108]. To increase effectiveness and efficiency, some researchers proposed combining multiple coverage criteria; Santelices et al. [85], showed that the cost of fault localization using combinations of coverage is less than using any individual coverage type, as different coverage features are good for different types of faults.

The use of coverage measures is based on the belief that a test suite that achieves higher coverage, or a stronger coverage measure, is more effective at exposing faults in the software under test. Some research [54] and real-world case studies [17] have borne out this belief. However, a large number of different test coverage measures have been proposed, and research is needed to study which are most effective.

Achieving a higher coverage after a certain level is very difficult except for some trivial programs due to cost overhead; to reduce the burden, some novel techniques like residual testing [75] and efficient path profiling [11] were introduced. There are some trade-off algorithms also proposed to minimize the use of resources (memory space and time) with a proportionally smaller decrease in fault detection [96]. Tikir et al. [93] proposed that instead of using static instrumentation, they would instead dynamically insert and remove instrumentation code, thus reducing the overall execution time for long-running program.

3.2 Instrumentation

Generally, software doesn't automatically produce coverage information; software needs to be instrumented to produce coverage-related data. Instrumented software can produce aggregated information, traces, or real-time feedback. It is worthwhile to mention that any kind of instrumentation imposes some form of performance and resource penalty. There are several ways to instrument software to get the coverage information [33] that include the following:

- **Source Code Instrumentation:** The original source code is annotated to provide coverage information without compromising the code's functional ability. One advantage of source-code instrumentation is that it has few problems with portability across different compilers of the same language. In our research we applied source code instrumentation; we developed tools that can instrument source code for C, C++ and Java to report branch coverage information. Please check Appendix A for more details.

- **Binary Instrumentation:** It is possible to instrument object or executable binary codes, especially important for situations where source code is not easily accessible [21]. The original source code or the language has little impact in this process.

- **Dynamic Instrumentation:** Without instrumenting the whole source or binary code of a program, Dynamic Instrumentation instruments a patch of code that needs to be observed while the program is running [19]. It reduces the burden of recompiling, linking and rerunning the whole process for large, time-consuming software.

There are two kinds of code coverage tools commonly used in the software industry: source code instrumentation such as IBM Rational Test RealTime, LDRA Testbed, IPL AdaTest or BullseyeCoverage and object code instrumentation such as VeroCel VeroCode or GreenHills GCover [27]. The reason for using object code instrumentation is that after compilation there may be some portion of object codes introduced that are not traceable from the source code. A typical example would be an array bounds checker which may be added during the compilation process and depends on the design of the compiler itself (that is, different versions or vendors). It is an extra measure so that safety critical software would not miss coverage information for introduced code.

3.3 Test coverage criteria in the software industry

Coverage measures boost confidence in a program's correctness [94], some form of structural testing is increasingly recommended to get a better understanding of testing thoroughness and its reliability. This is especially important for unit testing, which is by its nature a structural or white-box test [82]. Williams et al. [100] suggest in their study of Microsoft that a metric like percentage-of-code coverage (class, function, block and others) is a more reliable indicator than just mentioning the number of test cases.

There are some investigations related to test coverage effectiveness based on contemporary large industrial software. Mockus et al. [64] did some empirical analysis in two significantly large projects: Windows Vista, with its more than 40 million lines of code and a project at Avaya, with one million lines of code. They concluded that with the increase of test coverage, filed reported problems (that is, post-verification defects) decrease; but there is trade-off, as test effort increases exponentially with test coverage though reduction in field problems increases linearly with test coverage.

Rajan et al. [78] reported extensive use of some form of structural coverage criteria in avionics and other critical-system domains to measure the adequacy of test suites. They also reported some negative aspects of Modified Condition and Decision Coverage (MC/DC) [23], a widely used coverage criterion in those domains (in avionics MC/DC is considered as the standard [1, 45]); in their experiment, the MC/DC metric was highly sensitive to the structure of the implementation. Beside civil avionics, coverage analysis is also explicitly required for software in space systems (ECSS-40 standard) and nuclear systems (IEC-880) [27]. It is worthwhile to mention that none of those standards required any kind of data flow coverage. Jay et al. [95] report that the National Aeronautics and Space Administration (NASA) is using code coverage measures in their Mission Control Technologies (MCT) project. Matteo et al. describe a tool named Couverture that is currently used at AdaCore and Thales Aerospace (an avionics manufacturer which provides Air Data Inertial Reference Units (ADIRU) for the Airbus) [27].

The emergence of Agile as a mainstream software development method [95] which encourages the integration of development and testing [90] also ensures an extensive use of coverage measures at large software companies, including IBM and Microsoft [15]. Though a steady increase of structural coverage use in software industries has been reported, Smith and Williams report that there are some skeptics in the development community [88]; Runeson also reported the same in an earlier study [82].

3.4 Effectiveness and Coverage Measures

If coverage measure A subsumes coverage measure B, then a test suite that achieves 100% A coverage also achieves 100% B coverage. However, achieving the stronger coverage measure often involves more effort. If, when applied to actual programs, achieving 100% of A does not result in a test suite that is more effective (exposes more faults) than achieving 100% of B, then the effort of achieving 100% of A is not worth it.

Hutchins et al. [54] showed that test suites that achieved higher coverage were more effective, and that test suites that achieved high def-use coverage were more effective than those that achieved high branch coverage. To compare effectiveness, they hand-seeded faults into seven subject programs (the "Siemens programs") and measured the effectiveness of a test suite as the percentage of faulty versions that the test suite exposed. An earlier experiment by Frankl and Weiss [38, 36] also showed that "all-uses" (a form of DU) was significantly more effective than "all-edges" (Branch coverage) for five of their subject programs in terms of bug detection. These results raised the possibility that the effort of achieving measures stronger than branch coverage could pay off.

Hutchins et al.'s research left open the question of whether the test suites that achieved higher coverage or subsuming coverage measures were more effective because of some intrinsic quality of the coverage measure, or simply because the test suites they required were bigger (contained more test cases). Siami Namin and Andrews [87] found that both size of a test suite (number of test cases) and coverage were good predictors of effectiveness. They also found that a linear combination of *log*(size) and coverage yielded a good numerical prediction of effectiveness. They did not, however, compare coverage measures against each other.

Heimdahl et [47, 46] found that randomly generated test suites surprisingly perform better than those suites which were guided to achieve some structural certain kind coverage criterion(like decision coverage). They also found that in test suite reduction process test suite with similar level of coverage but less number of test cases drastically affects the fault finding capability.

3.5 Implementation of Def-Use Coverage

Hutchins et al.'s research suggests that def-use coverage is useful. Unfortunately, the practical implementation of def-use coverage poses several difficulties. The first is that it is apparently non-trivial to build a tool set for correctly recording and reporting def-use coverage. We were able to find only one working, compilable tool set that implemented def-use coverage for C, one for Java, and none for C++. The C tool (ATAC [49]) did not manage to successfully instrument and record coverage for two of the large C utility programs that we used as subjects. Recently Yang et al. reported on the coverage criteria measured by a wide range of commercial tools [105]; none of the tools surveyed implemented any form of def-use coverage. Although the source code analysis needed for def-use coverage instrumentation does not seem to be very complex, the above evidence suggests that its complexity is deceptive, and/or is high enough to discourage tool vendors from implementing it.

A second difficulty associated with def-use coverage is that the needed instrumentation slows down the system under test more than the instrumentation for controlflow-based coverage. For some systems this may make testing infeasible, and/or introduce timing bugs that do not appear in the uninstrumented system. Santelices and Harrold addressed this concern in their study of more efficient ways of implementing def-use coverage [83], but concluded that more work was needed to further improve the efficiency of def-use coverage.

A final difficulty concerns arrays. When the program under test contains arrays, the def-use tool implementor must decide whether to consider the whole array as one variable, or each array entry as a separate variable. If the whole array is considered one variable, then a def-use pair for the variable can be measured as executed even if one entry is assigned in the def and a different entry is used in the use. If each array entry is considered a separate variable, then the number of def-use pairs is greatly multiplied. In our research, 20 test cases for one of the subject programs (`jtopas`), when instrumented for def-use coverage, yielded bytecode files that were too big to be executed on the JVM; cutting down the size of an array in the code cured the problem, but resulted in tests that did not do the same thing.

3.6 MPSC

In response to the above issues, researchers seek coverage measures that are more predictive of test suite quality than branch coverage, but not as difficult or expensive to instrument or collect as def-use coverage. The candidate measures that we explore in this thesis are the “Multi-Point Stride Coverage”, or MPSC, measures defined above.

We had four main motivations for considering MPSC as a candidate coverage measure. First, it generalizes branch coverage, as noted above.

Second, because MPSC tracks sequences of branches, it may capture some of the same information that def-use coverage captures. Some def-use pairs connect statements in distant branches, similar to MPSC tuples with $g > 1$. It is easy to show that MPSC with a given value of g and p is not comparable to (neither subsumes nor is subsumed by) def-use coverage. However, the more practical question is whether, when applied to actual programs, MPSC is as predictive of high test suite quality as def-use coverage is. We address this question through experimentation, which we report on here.

Third, MPSC is relatively easy to instrument for, in widely-used procedural languages like C, C++ and Java. We built a simple source code transformation package, called JQXZ, that searches for `if`, `while`, `for` and `switch` statement constructs, and instruments the associated decisions and cases. We believe that similar simple transformations could be done on other source languages, and on bytecode and native code.

Fourth, we believed that the instrumentation for collecting MPSC data could be made efficient, by storing the data in a hash table of appropriate size and using an appropriate hash function. Our performance experiments, reported on in this thesis, bore out this belief.

Why did we not consider other established measures shown to subsume branch coverage, such as condition/decision, multi-condition, or Modified Condition/Decision Coverage (MC/DC)? The problem here is the surprising and important results of Rajan et al. [78], who showed that it is possible to subvert the intent of such coverage measures by restructuring programs. Rajan et al. showed that programmers can restructure a program into one which does the same thing, but whose 100% MC/DC coverage test suites are not as effective in exposing bugs. The same analysis holds for all three of the coverage measures mentioned above. We believe that MPSC will not be as vulnerable to source code transformation as MC/DC and related measures, because it is an interprocedural coverage measure defined in terms of dynamically-executed sequences of branches rather than the static structure of decisions. Studying this belief in more detail, however, is a subject for future work.

3.7 Antecedents to MPSC

We have not been able to find any mentions of measures equivalent to MPSC with $g > 1$ and $p > 1$. However, we do not claim any novelty for MPSC with $g = 1$. Measures similar to MPSC with $g = 1$ and $p > 1$ have appeared in the literature in the past, although curiously there has not been much research on them in recent years, and we believe that we are the first to investigate such measures empirically.

Pimont and Rault [77] defined a hierarchy of coverage measures equivalent to MPSC with $g = 1$. Chow [24] extended their work to coverage of state machine abstractions of programs by test cases, in particular "switch cover", which is similar to MPSC with $g = 1$ and $p = 2$. Woodward et al. [103] defined the notion of LCSAJ, or Linear Code Sequence and Jump, as a "body of code through which the flow of control may proceed sequentially and which is terminated by a jump". They also defined an infinite hierarchy of coverage criteria TER$_i$, for which TER$_{n+2}$ where $n \geq 1$ is coverage of distinct subpaths of length n LCSAJs. There is no discussion of whether the criterion is meant to be inter- or intraprocedural. Over 20 years later, Woodward and Hennell [104] showed that a coverage criterion equivalent to all (intraprocedural) LCSAJ sequences subsumes MC/DC under certain assumptions about program structure.

"Path coverage" is usually taken to mean coverage of all possible paths through a program. Even at the intraprocedural level, it has long been recognized that the number of such paths can be infinite [65] which is known as the path-explosion problem [84].

Zhu et al. [109] defined a *simple* path as one that does not repeat *edges*, while Ammann and Offutt [5] defined a simple path as one that does not repeat *nodes*, except that the first node can be the last (a refinement of Zhu et al.'s notion of "*elementary* path"). Simple path coverage under either definition does not subsume MPSC, since an MPSC tuple can contain a given branch an arbitrary number of times. However, MPSC with $g = 1$ and a high enough p does subsume simple path coverage.

Finally, Ntafos [67] defined *required k-tuples* coverage as coverage of all possible linked chains of k def-use pairs. In such a chain, a decision may be followed by another decision that is not the next to be executed, as in MPSC tuples with $g > 1$. The required k-tuples criterion presumably has the same implementation and efficiency problems as def-use coverage, which is a special case of it.

3.8 Mutation for software testing

Mutation testing is an error driven testing methodology, first introduced by Lipton as early as 1971 [58]. After four decades, mutation testing is still not very popular in the industry. According to a survey in 2000 done by Offutt and Untch [71], three reasons hinder the industrial adoption process: Industry assumes that mutation testing is stringent but not economically viable; there is a lack of enthusiasm towards integrating unit testing (though it has changed dramatically in last the decade); and there is a lack of automation to support mutation analysis and testing. There are some techniques proposed to get some control on cost that include Mutant Reduction and Execution Cost Reduction [58]:

- **Mutant Reduction:** Mutant reduction aims to reduce the number of mutants with a minimal decrease in effectiveness. As each mutant incurs a significant cost for analysis and execution, reduction strategies that include sampling [102], clustering [56], ordering [57] and selective mutation [70] can help create a smaller set of mutants with equivalent effectiveness. In our work we use a form of sampling technique to reduce the number of mutants.

- **Execution Cost Reduction:** It is possible to reduce mutant-execution cost through some optimization process. One such approach is to use weak mutation instead of strong mutation. In a strong mutation, the fate of a mutant is decided after the complete execution and comparison of outputs; in a weak mutation, any change of execution in the mutated block is considered as capturing the mutant. Empirical studies suggest weak mutation is less computationally expensive than strong mutation [39, 62]. There is also some discussion of using advanced computers to distribute computation load into multiple processors [69].

Though industrial adoption of mutation testing is not yet widespread, the field has become more mature in the last decade. A significant amount of research has been done to make it cost effective and economically viable [58].

We used mutation analysis to compare our proposed coverage criteria (MPSC) to data flow coverage criteria. In the following sections, we will discuss some general concepts related to mutation. Finally, we will discuss with the aid of a flow chart how we use mutation analysis in our research.

3.8.1 Important concepts related to mutation

A mutant of a program is generated through some induced faults applied to the original program. The mutation generation process creates many versions of a program, each containing a single fault [71]. Here are some important concepts related to mutation:

- **Mutation Operator:** This consists of rules that, applied to a program, would change it and create mutants. Some common mutant operators include sign replacement in a statement and changing values assigned to a variable. In Table 3.1 we show the mutation operators [7] we used in our experiment.

Table 3.1: List of Mutant Operators we used in our research

1	Replace an integer constant C by 0, 1, -1, ((C) +1), or ((C) - 1).
2	Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class.
3	Negate the decision in an if or while statement.
4	Delete a statement.

- **Mutant Killing:** A mutant is considered killed or captured when it produces a different output than the original program for at least a single test case.

- **Equivalent Mutant:** When a mutant produces the same output as the original program for any test, it is considered an equivalent mutant. It is an undecidable problem to prove mutant equivalency; in general, a mutant is considered equivalent after executing all given tests successfully without getting any output difference.

- **Infeasible Mutant:** There are some mutants which take too much time to execute or produce any results. We considered them as infeasible mutants, and they were discarded from mutant pools.

Figure 3.1 depicts a generic mutation testing approach. This approach tries to divide mutants into two classes: killed mutants and equivalent mutants. After analyzing equivalent mutants, the test suite can be improved to a certain degree and the process restarted.

3.8.2 Why it is acceptable

There are two hypotheses behind mutation that allow us to claim it as a viable testing strategy. They are as follows:

- **Competent Programmer Hypothesis:** DeMillo et al. [28] first introduced this concept. It assumes that programmers are competent, so they know what they are developing and their software would be almost correct. There is still a chance of having faults in the program but they would not be major ones and some syntactical changes would correct those faults. A detailed theoretical discussion can be found in Budd et al. [20].

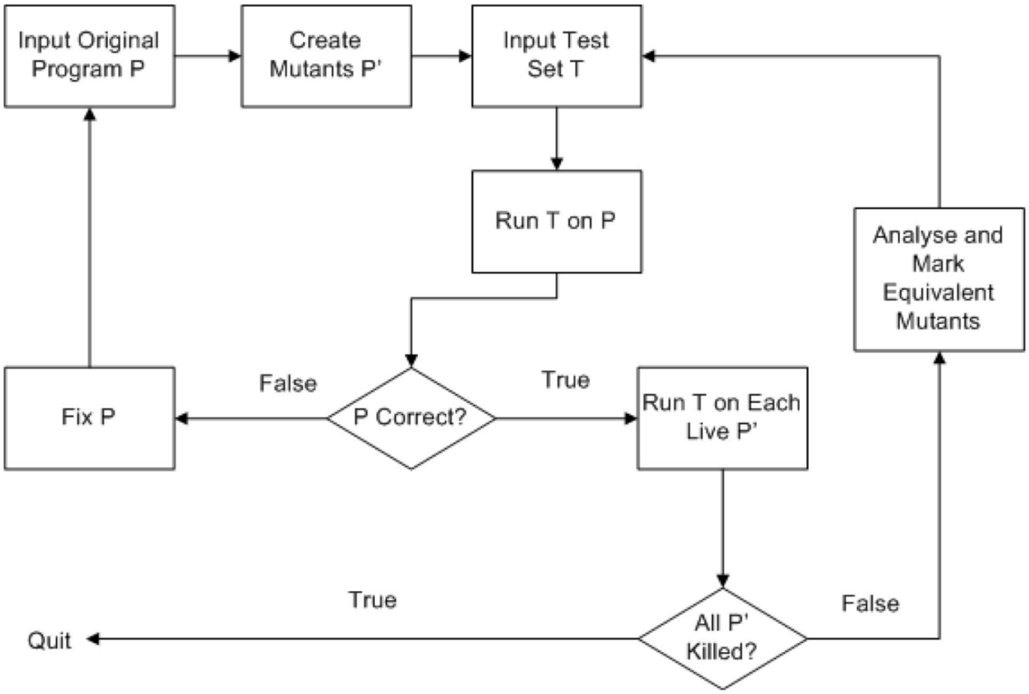


Figure 3.1: Mutation analysis model [71]

- **Coupling effect:** DeMillo also proposed coupling effects in the same paper [28]. If a test suite is sensitive enough to detect simple errors (induced faults) then it should able to capture the more complex errors. In a more formal way Offutt [68] describes the hypothesis as follows : “Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults”.

Empirical research has shown that mutation testing is valid for software testing. Andrews et al. [6] suggest that mutants are similar to real-life faults but may not very similar to hard hand seeded faults which are sometimes used in testing research.

3.8.3 How we use it

In our research, we follow Hutchins et al.’s general experimental design, using mutation [6, 7] to generate faulty versions automatically. We then compare MPSC to def-use coverage in order to measure how well each predicts the effectiveness of a test suite, when test suite size is taken into account.

We used “mutgen” developed by Andrews et al. [6] to generate mutants. It was designed to work with any C, C++ or Java program, and it was written in Prolog.

We have created a database for each subject program under test containing all possible mutants generated by mutant operators described in Table 3.1 using mutgen. We chose mutants randomly from the database, ensuring that a single mutant was chosen only one time. If the mutant was killed after running the entire test suite, that mutant was accepted for the purposes of comparison. We also recorded related test case information for comparative analysis. If a mutant takes a much longer time to execute than the original, we discarded it as infeasible. Then we did the same process until we had 100 mutants for each subject program.

We show our mutation analysis process in a flow graph in Figure 3.2.

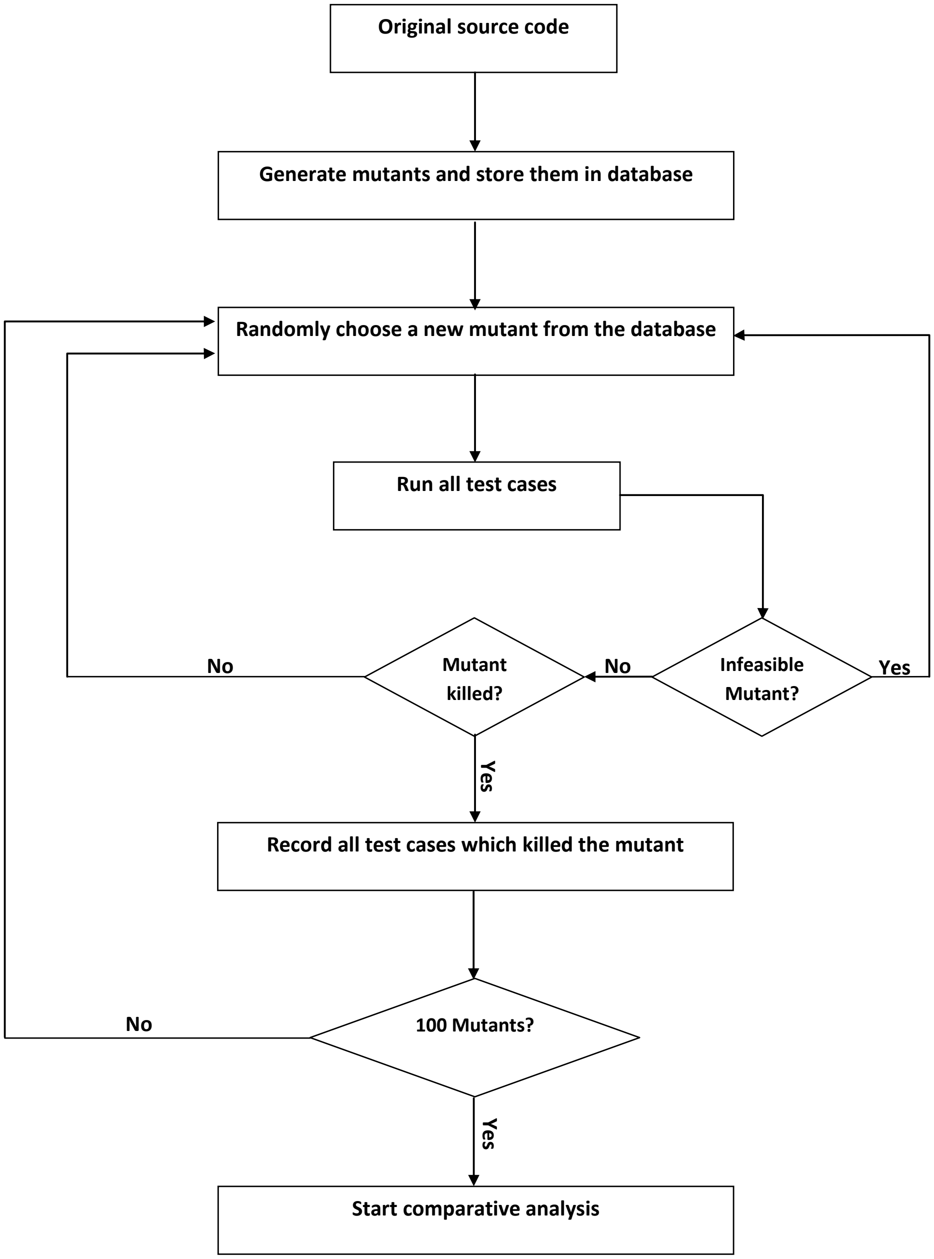

Original source code
Generate mutants and store them in database
Randomly choose a new mutant from the database
Run all test cases
Mutant killed?
No
No
Infeasible Mutant?
Yes
Yes
Record all test cases which killed the mutant
100 Mutants?
No
Yes
Start comparative analysis

Figure 3.2: Comparative mutation analysis for RQ3

Chapter 4

Basic Properties of MPSC

Our first research question (RQ1 in chapter 1) deals with the basic properties of MPSC: "How many MPSC tuples typically need to be collected for a program, and how is that number related to other program metrics?"

Data structures for collecting MPSC data, for instance hash tables, can be made more efficient if we can approximate how many entries will be needed. The upper bound for this number is the measure defined in chapter 2, $\mathit{maxcov}(S, g, p)$ for subject program S, gap g and number of points p, since this is the size of the set of all (g, p)-tuples that can possibly be collected for S.

Because MPSC with $g = 0$ or $p = 1$ is equivalent to branch coverage, an upper bound for $\mathit{maxcov}(S, 0, p)$ for any p can be determined statically, by simply counting the number of branches in the program. We therefore wanted to see if there is a relationship between $\mathit{maxcov}(S, 0, p)$ and $\mathit{maxcov}(S, g, p)$ for general g and p. Because $\mathit{maxcov}(S, 0, p)$ is the same for any p, and $p = 2$ was the smallest p for which we collected data, we use $\mathit{maxcov}(S, 0, 2)$ in this thesis.

We use $\mathit{maxcovratio}(S, g, p)$ to mean the ratio $\mathit{maxcov}(S, g, p)/\mathit{maxcov}(S, 0, 2)$. We can think of this as the memory multiplication factor needed to collect MPSC coverage at gap g and p points.

Table 4.1: List of subject programs.

	Program	Type	Lang.	SLOC	# Test Cases
1	`concordance`	Text processor	C++	1492	372
2	`flex`	Lexer generator	C	10459	567
3	`gzip`	Data compression	C	5680	211
4	`grep`	String matching	C	10068	809
5	`sed`	Stream editor	C	14427	370
6	`printtokens`	Lexical Analyzer	C	726	4130
7	`printtokens2`	Lexical Analyzer	C	570	4115
8	`replace`	Pattern matcher	C	564	5542
9	`schedule`	Data structure	C	412	2650
10	`schedule2`	Data structure	C	374	2710
11	`tcas`	Hardware control	C	173	1608
12	`totinfo`	Information measure	C	565	1051
13	`jtopas`	Tokenizer library	Java	5400	207
14	`nanoxml`	XML parser	Java	7646	216
15	`xml-security`	Security library	Java	16800	84

4.1 Subject Programs

We addressed our research questions, and in particular RQ1, by collecting data on 15 subject programs in three languages (C, C++ and Java). We obtained the subject programs and corresponding test pools from SIR, the Software-artifact Infrastructure Repository (`http://sir.unl.edu/`), except `concordance`, which was converted to a subject program at our university [4]. There are eleven C, three Java and one C++ programs used in our research. We chose these programs as they are used in similar research and accepted as standard. In Table 4.1 we give the subject programs' names and relevant information collected from SIR (detail description of those programs given in Appendix B).

We included the small Siemens programs in our research (programs 6-12) because of the thoroughness and size of their pools of test cases. In Table 4.2, we quantify the benefits this thoroughness and size give us. The column "Branch Cov" gives the source branch coverage achieved by the entire test pool of each subject program. The Siemens program test pools are the ones with the highest coverage; in addition, Rothermel et al. [81] state that the pools cover

Table 4.2: Additional information of the subject programs. Branch Cov: branch coverage of whole test pool. TSD: test suite diversity measure. PEPC: partial execution pattern constant. ALR: Ratio of average ncov per test case and SLOC

	Program	Branch Cov (%)	TSD(50) (%)	PEPC (C)	ALR(1,2)	ALR(10,5)
1	`concordance`	84.4	86.6	1.41	0.0509	0.6734
2	`flex`	73.3	91.2	1.26	0.0942	0.9873
3	`gzip`	48.3	76.3	1.46	0.0676	13.0052
4	`grep`	44.6	93.8	1.43	0.0455	0.9266
5	`sed`	34.9	86.5	1.13	0.0462	0.2039
6	`printtokens`	91.5	98.8	1.46	0.0879	0.5023
7	`printtokens2`	95.8	98.8	1.07	0.1732	0.7321
8	`replace`	97.6	99.1	1.85	0.1219	0.4533
9	`schedule`	94.6	98.1	2.34	0.1414	0.9433
10	`schedule2`	91.8	98.2	2.04	0.2106	2.1069
11	`tcas`	86.7	96.9	0.53	0.0384	0.1574
12	`totinfo`	87.5	95.2	1.50	0.1002	0.5676
13	`jtopas`	71.3	75.8	1.21	0.0376	0.3907
14	`nanoxml`	69.1	76.9	0.98	0.0302	0.1592
15	`xml-security`	36.8	40.5	1.16	0.0179	0.1627

every *feasible* branch at least 30 times.

Larger test pools allow experimenters to select experimental test *suites* that differ more from each other (are more diverse). Larger test suite diversity allows us to take a more representative sample from the space of possible test suites for the program. To quantify this test suite diversity, we define TSD(x) as the expected fraction of the test cases of one randomly-chosen test suite of size x that would not appear in another randomly-chosen test suite of size x. (This is equivalent to $1 - (x/n)$, where n is the size of the test pool.) In Table 4.2 we give the value of TSD(50) for each program, since the largest random test suites that we generated were of size 50. Again the Siemens programs show the highest values of test suite diversity. [1]

Despite the lower coverage and/or lower test suite diversity of the other subject programs, we retained them in our set of subject programs because they are larger and represent realistic situations in which the available test suites achieve lower than maximal feasible coverage.

[1] Next column PEPC will be discussed in Section 4.3.2

Another factor that may affect the effectiveness of a test suite is the average coverage of the test cases in the test pool. We calculated the average *ncov* for test cases for each program and for for $g = 1, p = 2$ and $g = 10, p = 5$. Different sizes of programs will have test cases with different average *ncov*, so we scale the average *ncov* by dividing it by the numbers of source lines of code (SLOC) in the program. We call the result ALR for "Average *ncov*/ Lines of code Ratio"; it is shown in the last two columns of table 4.2. There is no noticeable pattern to the values of ALR, indicating that this unlikely to affect our results.

4.2 Procedure

We performed the entire experiment in the UNIX environment. We instrumented the programs using our instrumentation tool (JQXZ) so that they recorded to a disk file the full sequence of branches followed. In Figure 4.1, we show the process of using JQXZ to collect traces. Key steps are as follows:

- **Instrumentation:** A non-instrumented version of the subject program was modified by JQXZ to make it ready to provide branch traversal information in the execution phase. It modified branch statements only, by inserting a function call and adding related references at the beginning of the source file (for example, import JQXZ class in Java). We also had to modify test-drivers for two of our Java programs `jtopas` and `xml-security` to ensure that individual test cases produce separate trace files.

- **Test script generation:** SIR provided a standard test database for each of the subject programs. To get the test script from the database we used a tool named MTS ("Make Test Script"), also available from SIR.

- **Trace Collection:** After compilation of the instrumented source code, the software was ready to generate branch execution traces. We ran the test script and collected trace files for each of the test cases.

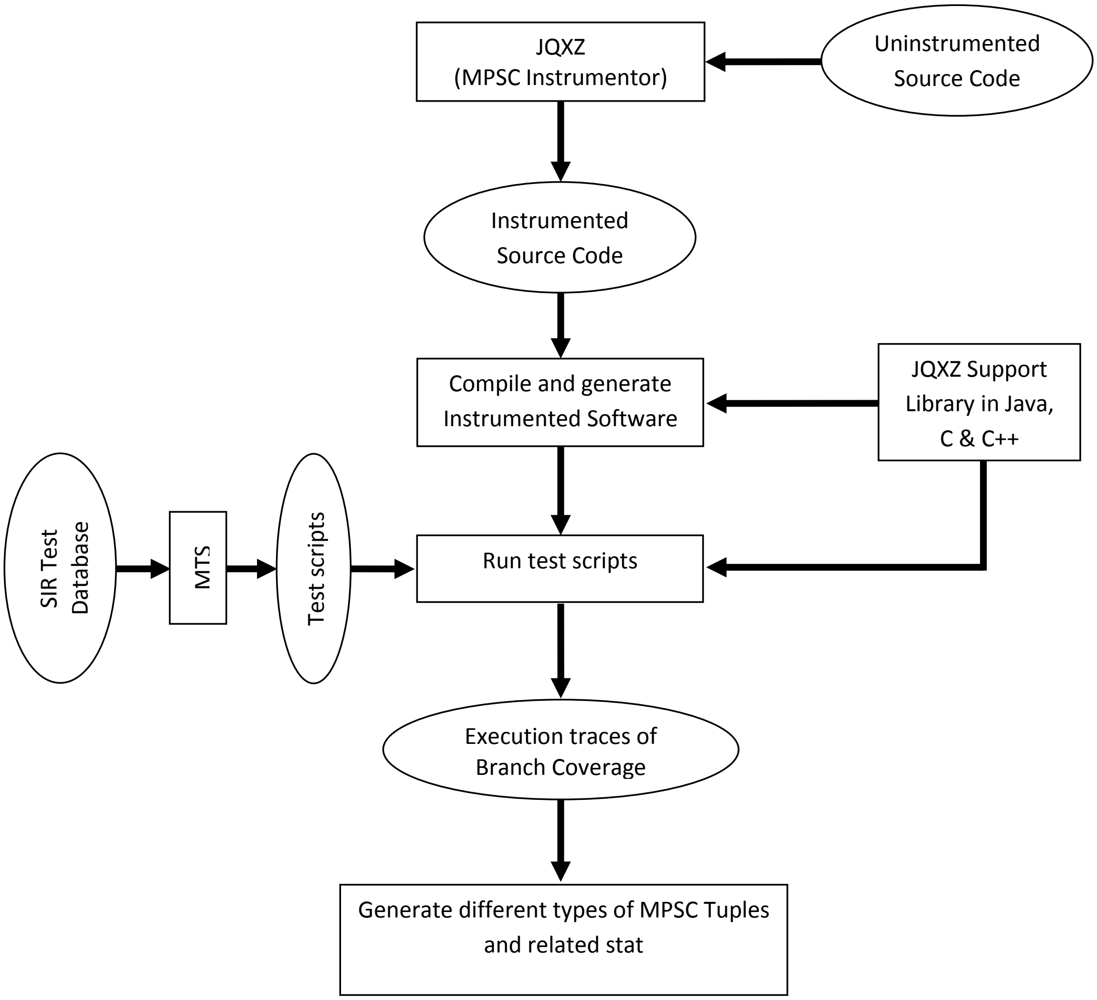

JQXZ
(MPSC Instrumentor)
Uninstrumented Source Code
Instrumented Source Code
Compile and generate Instrumented Software
JQXZ Support Library in Java, C & C++
SIR Test Database
MTS
Test scripts
Run test scripts
Execution traces of Branch Coverage
Generate different types of MPSC Tuples and related stat

Figure 4.1: Procedure to collect MPSC using JQXZ

- **MPSC Collection:** We then processed the individual trace files to extract all the MPSC tuples executed by each test case, for $g = 0$ to 10 and for $p = 2$ to 5. We chose this range of values based on what we expected would be feasible to record. We computed the value of $maxcov(S, g, p)$ for each subject program S and each value of g and p.

We then illustrated the results using graphs and investigated the relationships between g, p, and *maxcov* using linear regression. We will describe in detail the presentation and analysis of MPSC in the next sections.

4.3 Results and Analysis

We observed the behavior of MPSC using some representational graphs which revealed some intrinsic relationship between independent variables g, p and dependent variables such as *maxcov*. Encouraged by our preliminary observations, we applied regression to get a linear model which can characterize MPSC and help to get the upper bound of the number of MPSC tuples. In the following sections we will describe our analysis.

4.3.1 Early Observations

First we try to understand the nature of MPSC for individual test cases. In Figure 4.2 and 4.3 we show the *ncov* of individual test cases in a box-and-whisker plot. There is no distinguishable pattern for different g and p across different subject programs or languages. There are many outliers, and the median slowly increases with the increment of g and p. These developments were expected; as individual test cases are very different from each other, so too is their coverage information. Generally we can see a clearer pattern if we consider the data in a more accumulative form such as the total coverage information of a test suite, in our case *maxcov*.

Figure 4.4 and 4.6 show the relationship between g, p, and *maxcov*, for the subject programs `replace(C)`, `concordance(C++)` and `nanoxml(Java)`. As expected, *maxcov* increases with increasing g and p. For other programs irrespective of the languages, the lines are straighter

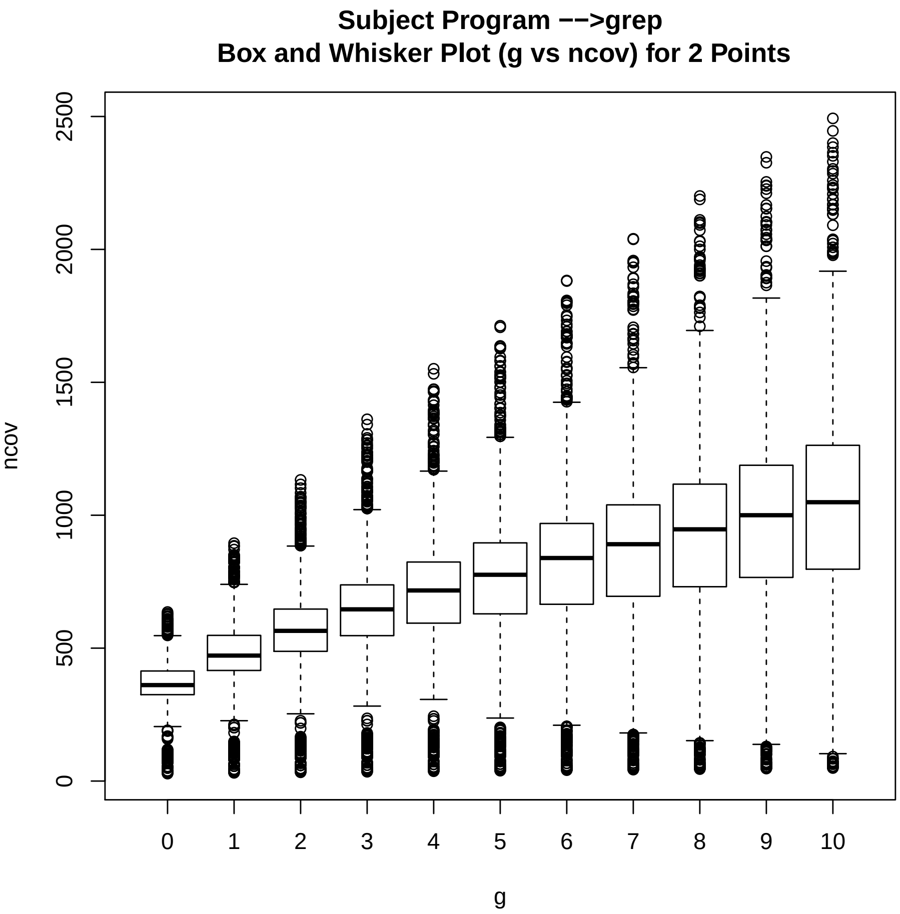


Figure 4.2: Box and Whisker plot (*g* vs *ncov*) for individual test cases, 2 Points MPSC, subject program `grep`

Note: The thick horizontal line inside the box indicates the median of all the points in the column. 25% of the points are below the bottom of the box and 25% of the points are above the top of the box. The distance between top and bottom of the box is called the interquartile distance. The small circles are outliers. Outliers are points that are more than 1.5 times the interquartile distance from the top or bottom of the box. The whiskers extended above and below the box to the last non-outliers data point.

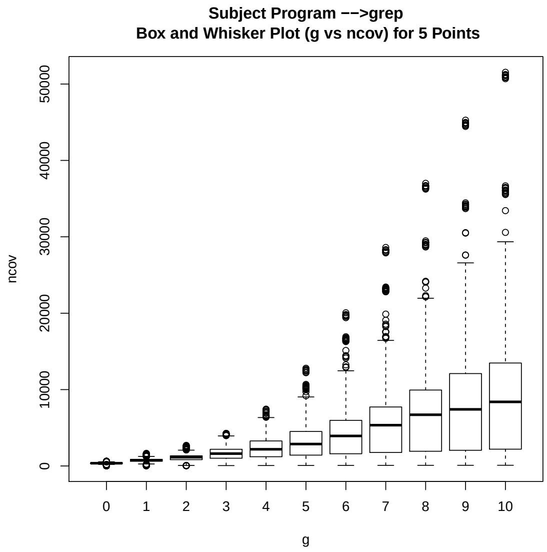


Figure 4.3: Box and Whisker plot (*g* vs *ncov*) for individual test cases, 5 Points MPSC, subject program `grep`

or concave-down rather than concave-up, but the relationships between the lines are similar. We can observe similar kinds of patterns if we replace *maxcov* with *averagencov*, as shown in Figure 4.7.

Patterns are different for the program `tcas`. For instance, *maxcov* increases with increasing g and p only for points with $g \leq 5$ (See Figure 4.5). It is a very small program (the smallest program in our experiment) and contains no loop structure. There are only a small number of unique paths possible in `tcas`, and each of the paths is short, producing anomalous results when $g > 5$. So for `tcas` we omit all points with $g > 5$ for further analysis.

Our early observations suggested that there may be a relationship among independent variables g and p and dependent variables such as *averagencov* and *maxcov* for each of the programs. So we tried to fit lines (linear, quadratic and cubic) to the points, using the process called regression, which finds a line that fits the points best. Some early results (see Figure 4.8 and 4.9) also suggested that it is possible to get mathematical models for different combinations of g and p, but we were looking for a generic model that combines all g and p. For further observation we needed to stabilize the data [55], so we applied log on g, p and *maxcov*

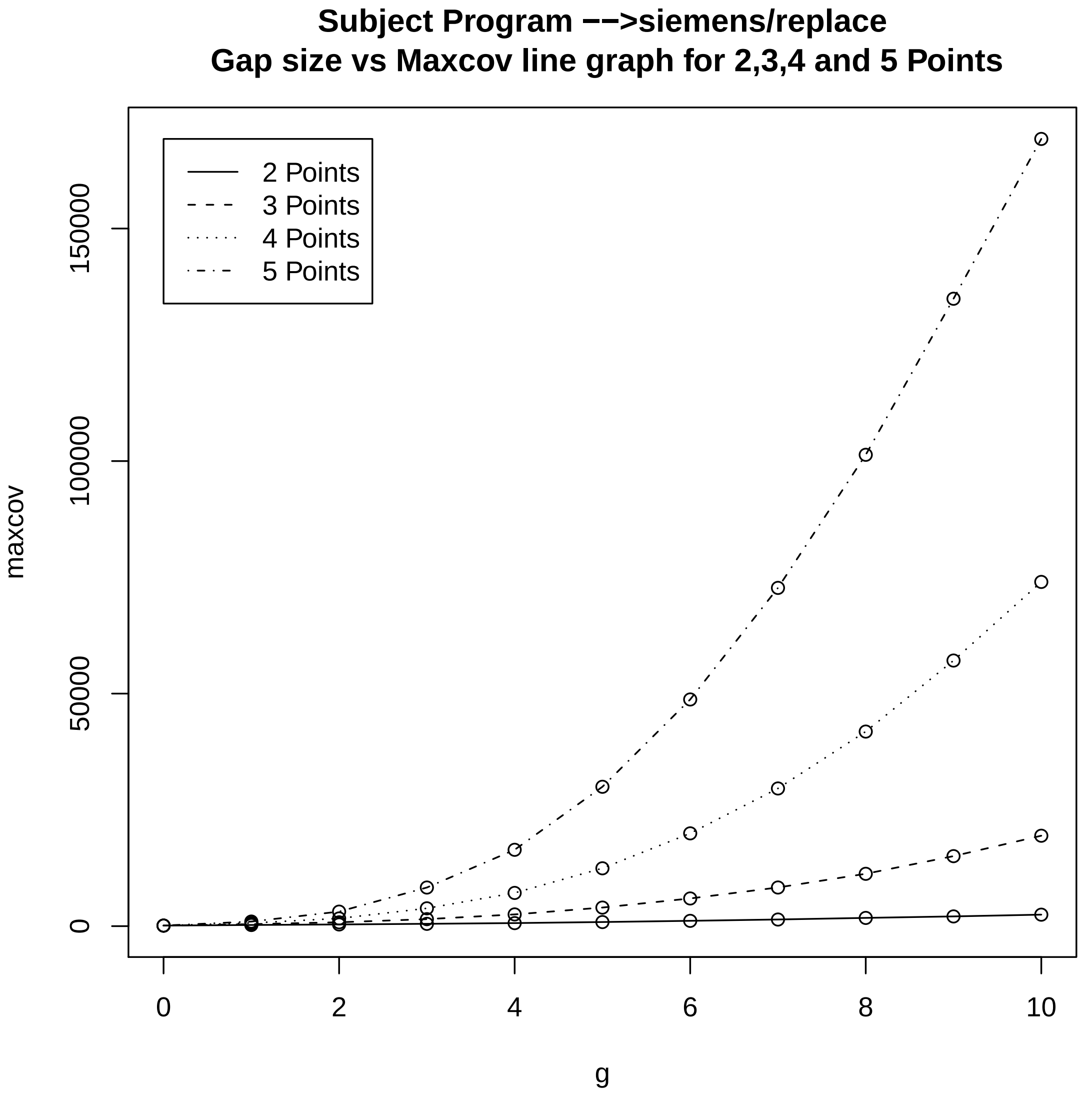


Figure 4.4: Graph of maxcov for given values of *g* and *p*, for subject program `replace`, a C program

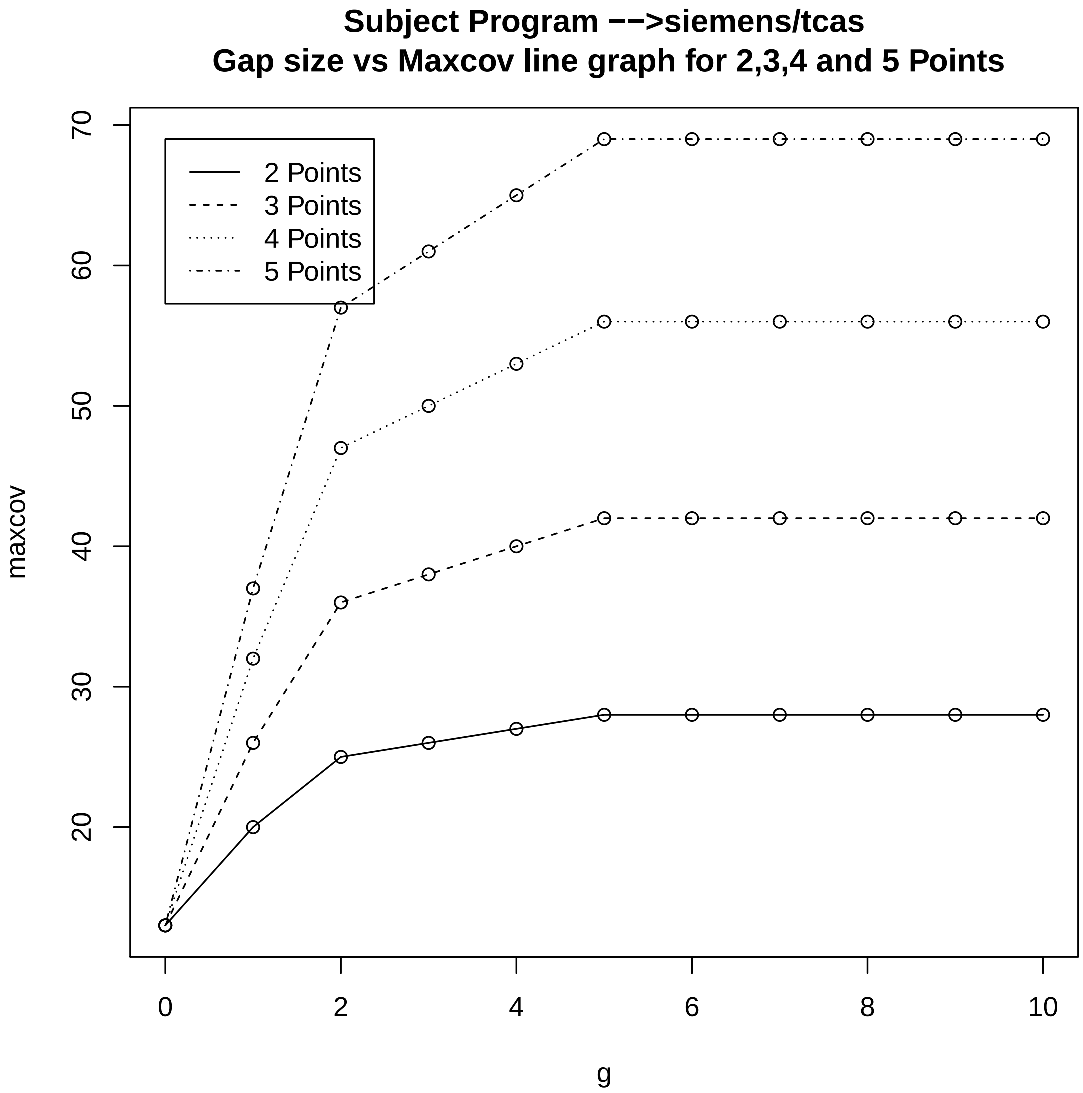


Figure 4.5: Graph of maxcov for given values of *g* and *p*, for subject program `tcas`, a C program

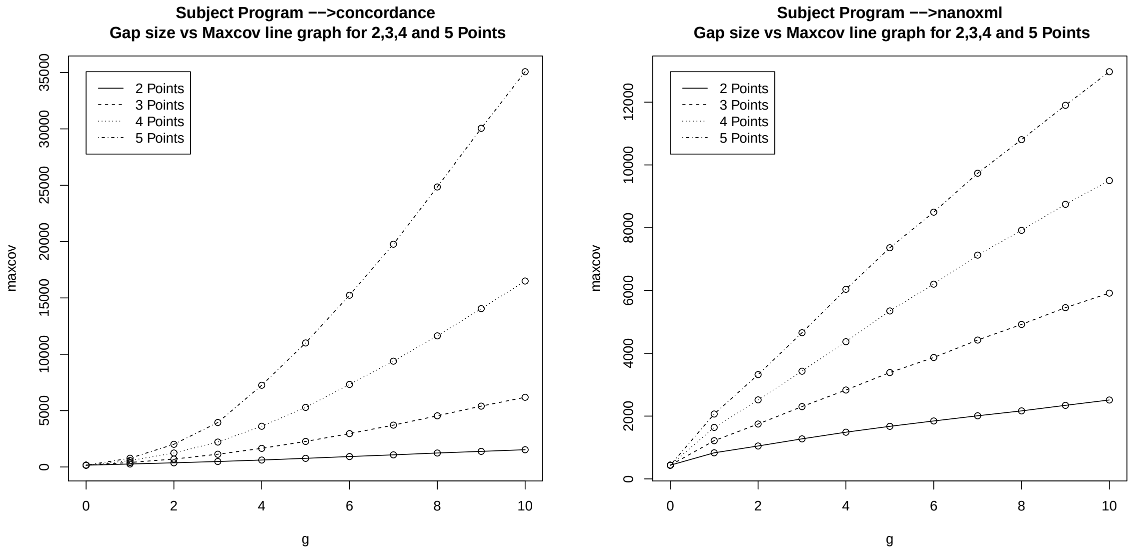


Figure 4.6: Graph of maxcov for given values of *g* and *p*, for subject programs `concordance` (C++) and `nanoxml` (Java)

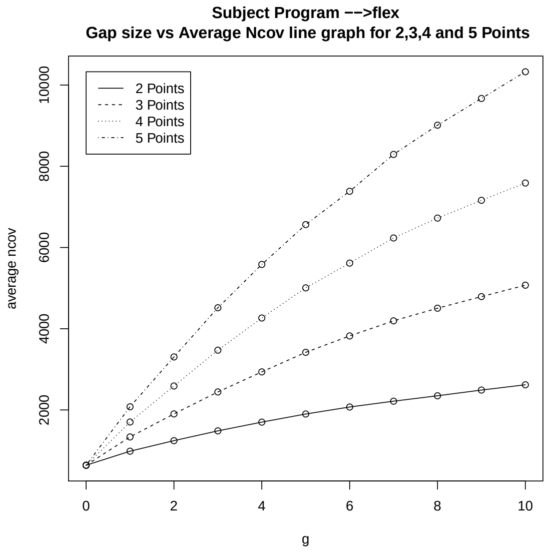


Figure 4.7: Graph of average *ncov* for given values of *g* and *p*, for subject program `flex` (C)

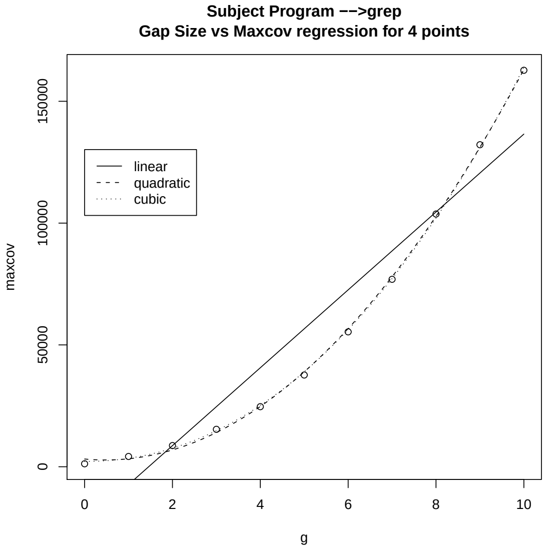


Figure 4.8: Regression *maxcov* vs *g* where $p = 4$, for subject program `grep`

and then combined the two independent variables g, p through multiplication. Because g can be 0, so we added 1 to g before taking the log. In Figure 4.10 we show the resulting data for `apache-xml-security` in a scatter plot[2]; it is obvious from the figure that a linear model may exist to reveal their relationship[3]. The results for the other programs are similar.

4.3.2 A mathematical model of MPSC

Using the statistical package R [97], we searched for linear relationships between g, p, and *maxcov*. We find that the following linear model can be used to describe MPSC for a subject program:

$$log(maxcovratio(S, g, p)) = C * log(g + 1) * log(p) \tag{4.1}$$

[2] In all scatter plots, the straight green line shows the best linear fit, and the curved line shows the LOWESS (locally-weighted best fit) smoothing line.

[3] We ignore *averagencov* for further analysis as its behavior is similar to *maxcov*: basically, *averagencov* is a scaled-down version of *maxcov*.

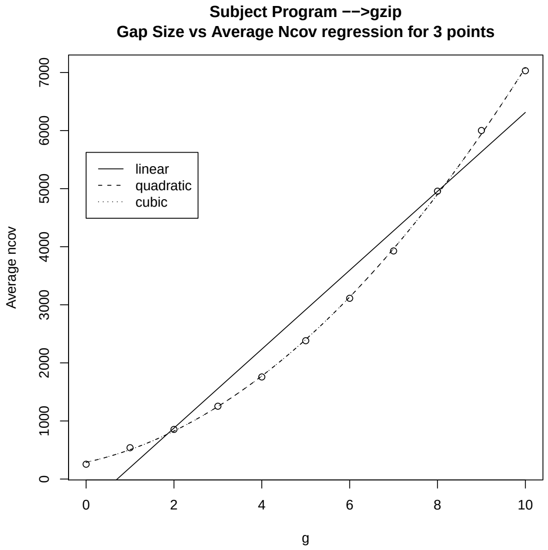


Figure 4.9: Regression *average ncov* vs g where $p = 3$, for subject program `gzip`

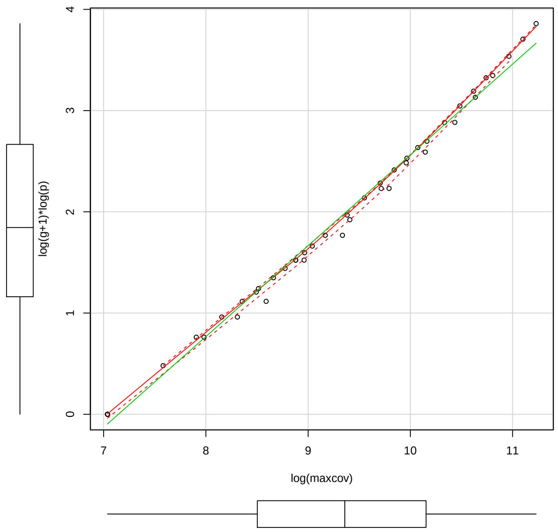


Figure 4.10: Graph of log(maxcov) vs “$\log(g + 1) * \log(p)$” , for subject program `apache xml security`

where C is a constant for the subject program S, which we call the "partial execution pattern constant". We find that this model works for all of our subject programs with high accuracy[4]. Here we use *macovratio* over *maxcov*, as through regression analysis we found that *maxcov* of $g = 0$ was a constant. It also suggests that there is a direct relationship between higher orders of MPSC and basic MPSC or branch coverage.

Interestingly, each program has its own partial execution pattern constant C. The value of C for each program is given in the last column of Table 4.2. This constant does not seem closely related to any other program metrics.

4.3.3 Searching for a comprehensive model

We attempted to find a model for C that worked for all of our subject programs, taking into account source lines of code (SLOC) and number of modules (NOM). In Table 4.3 we show the result; column 2 contains different formulas and column 3 contains the corresponding adjusted R^2 value[5]. Before applying those formulas, we combined MPSC data from all subject programs with the corresponding LOC and NOM values of each program; after that we applied linear regression using each of the formulas. Table 4.3 shows that none of those formulas worked. SLOC and number of modules did not have a statistically significant relationship to C. For instance, the smallest programs (the Siemens programs, which also have the highest test pool coverage) include the program with the highest value of C and the program with the lowest value of C. So our quest for a universal C value that may work across different programs was not fulfilled. This may indicate the complex nature of any program, where similar structural metrics do not produce similar execution patterns.

In Figure 4.11, we show the actual value of $log(maxcovratio)$ compared to the value predicted by the model, for all values of g and p and all subject programs, where each program's

[4] The adjusted R^2 value, a measure of the predictive accuracy of a linear model, is greater than 0.96 for all programs and greater than 0.99 for most. The predicted coefficients have high statistical significance, with $p < 0.001$ for all programs.

[5] Here X ~ A * B means we are looking for linear model of the form X $\cong$ C_1 + C_2 * A + C_3 * B.

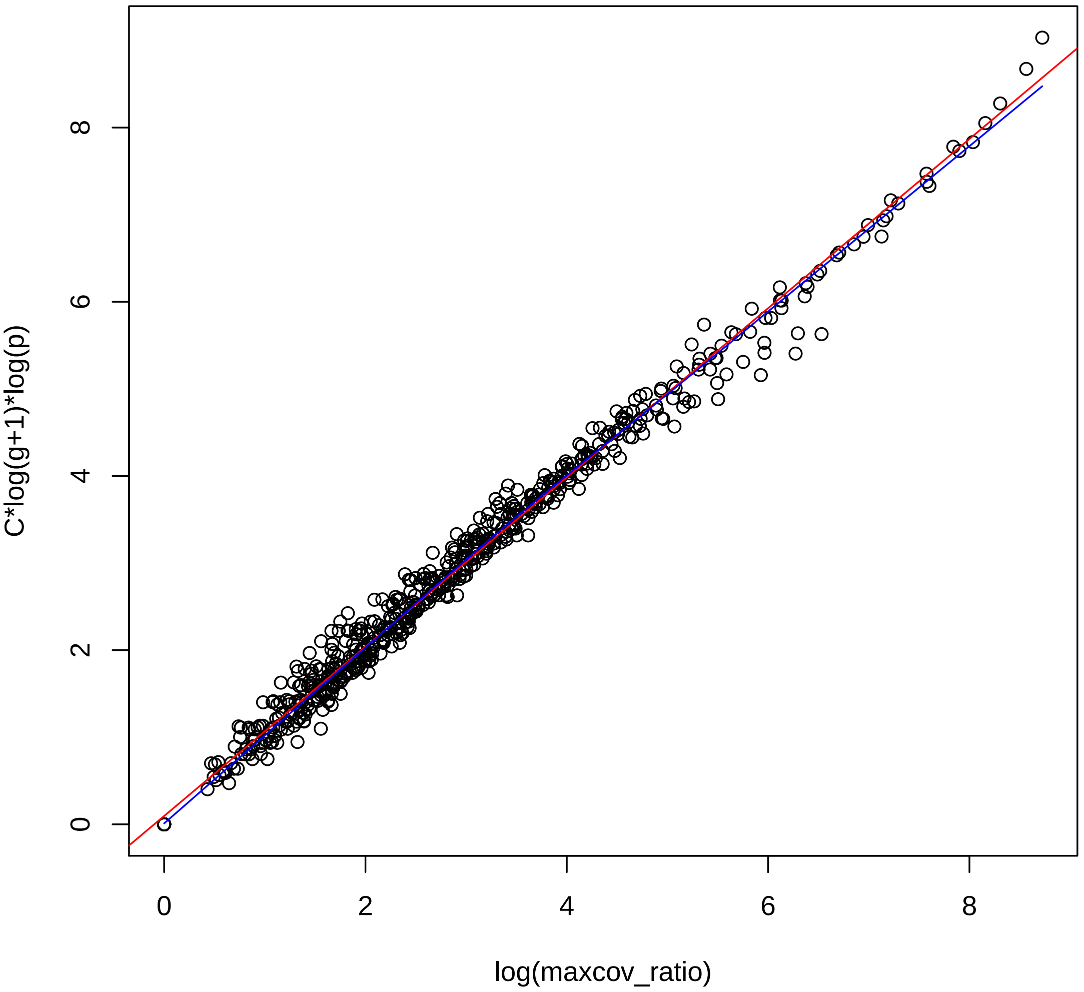


Figure 4.11: Actual vs. Predicted values of *log*(*maxcovratio*) for all programs.

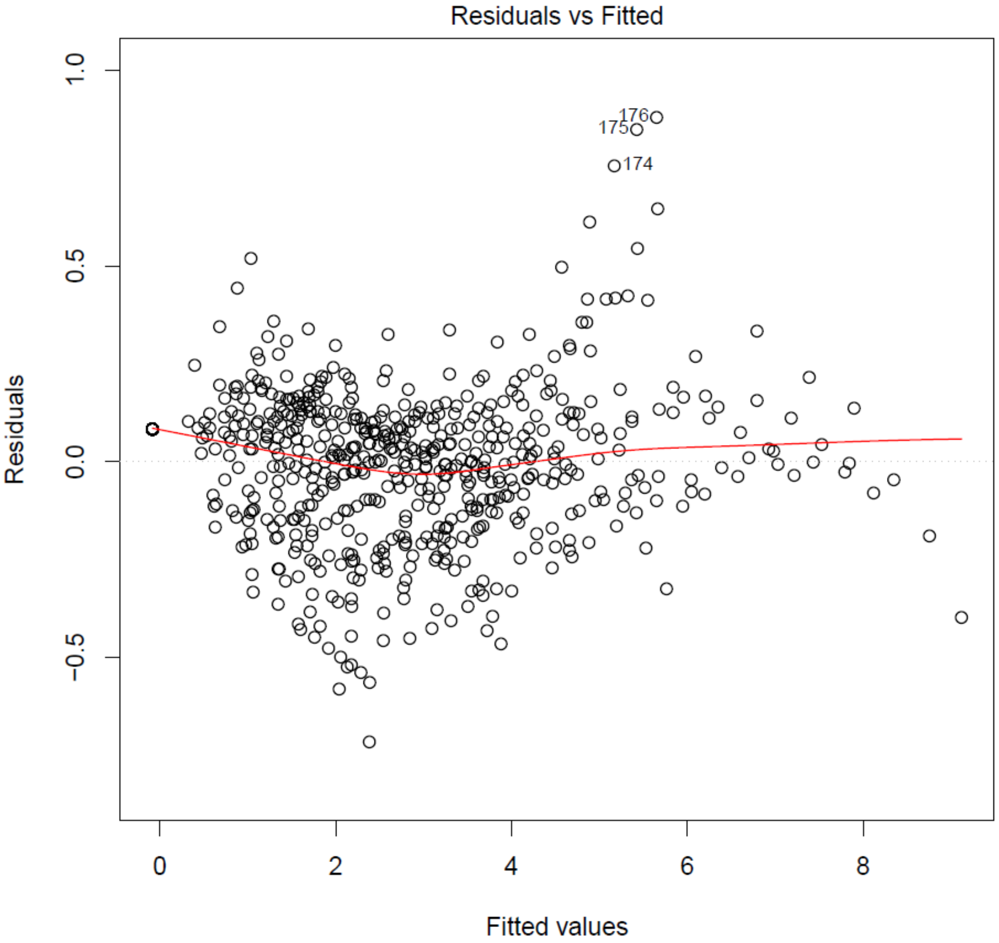


Figure 4.12: Residual plot of Actual vs. Predicted values of *log*(*maxcovratio*) for all programs.

Table 4.3: List of Generic Equations

	Tested Formulas	Adjusted R-squared
1	log(maxcovratio(S, g, p)) ~ log (g+1) * log (p)	0.7715
2	log(maxcovratio(S, g, p)) ~ log(g) * log(p) * log(LOC)	0.5183
3	log(maxcovratio(S, g, p)) ~ log(g) * log(p) * log(NOM)	0.4166
4	log(maxcovratio(S, g, p)) ~ log(g) * log(p) * log(LOC) * log(NOM)	0.2331
5	log(maxcovratio(S, g, p)) ~ log(g) * log(p) + log(LOC)	0.1174
6	log(maxcovratio(S, g, p)) ~ log(g) * log(p) + log(NOM)	0.2449
7	log(maxcovratio(S, g, p)) ~ log(g) * log(p) + log(LOC) * log(NOM)	0.003763
8	log(maxcovratio(S, g, p)) ~ log(g) * log(p) + (log(LOC) / log(NOM))	0.4847
9	log(maxcovratio(S, g, p)) ~ log(g) * log(p) / (log(LOC) * log(NOM))	0.5205
10	log(maxcovratio(S, g, p)) ~ log(g) * log(p) / log(LOC)	0.8548
11	log(maxcovratio(S, g, p)) ~ log(g) * log(p) / log(NOM)	0.5361

predicted value is based on its own partial execution pattern constant *C*. Figure 4.12 shows the residual vs. fitted plot of the model: the randomness of residual values suggests the soundness [34] of the model.

4.3.4 Practical approach to predict MPSC

There are practical uses to knowing the maximum number of probable MPSC tuples, such as memory management in the testing process. In most cases path or sub-path count is infeasible due to the path-explosion problem, which may produce an infinite number of paths even for a small program. With MPSC it is possible through rigorous static analysis of the source code to get the number of tuples for specific *g* and *p*, but the complexity and overhead discourage it.

An alternative approach would be to simplify equation 4.1 to predict the value of *maxcov* as follows:

$$maxcov(S, g, p) \cong maxcov(S, 0, 2) * p^{(C*log(g+1))} \quad (4.2)$$

Using the above equations, we can estimate the number of MPSC tuples that typically need to be collected for a program. As noted in chapter 2, $maxcov(S,0,2)$ is equivalent to the number of branches in the program, which can be determined by simple static analysis [6]. To estimate C, we can use the highest value of C observed so far for any program, which is 2.34. A hypothetical MPSC tool could obtain an upper bound for $maxcov(S,0,2)$ using source code analysis and then use equation 4.1 to find an upper bound for C.

4.4 Summary

- Collecting different types of MPSC is straight-forward. We developed a simple tool JQXZ for that, but any commercial tool that preserves the sequence during the trace-collection process can also be used.

- MPSC showed some interesting properties, such as the size of the MPSC sets follows a mathematical pattern.

- We found a new metric, the "partial execution pattern constant", which is unique for each program and seems unrelated to other traditional software metrics.

- We did not find a universal model to generalize the calculation of *maxcov* across all subject programs, which may suggest the uniqueness of the execution pattern of individual programs.

- It is possible to predict the size (upper bound) of the MPSC set using its characteristic equation, which is convenient compared to rigorous static analysis.

[6] Counting the number of loops and conditional statements is straight-forward.

Chapter 5

MPSC and Def-Use

Research questions RQ2 and RQ3 have to do with the relationship between MPSC and def-use coverage. RQ2 asks whether MPSC with some value of *g* and *p* is similar to def-use, in order to answer the direct question of whether MPSC can be used in place of def-use. RQ3 asks a question which is more relevant to software engineers considering the use of coverage criteria: whether MPSC is as predictive as def-use of the quality of a test suite.

5.1 Data Collection

We instrumented as many of our subject programs as possible with tools that measured def-use coverage. We used ATAC [49, 50] to instrument the C programs; all seven of the Siemens programs and two of the large Unix utilities (`flex` and `grep`) could be instrumented successfully. ATAC could not successfully instrument the C++ program (`concordance`) or the remaining two Unix utilities (`gzip` and `sed`). We modified ATAC slightly in order to print unique identifiers for each of the def-use pairs covered by the program. We obtained Santelices' tool DUAF [83] and instrumented our three Java subject programs with it.

We then ran each test case on each of the instrumented programs, and recorded which def-use pairs were covered by which test cases; this supplemented the information we collected earlier (see chapter 4) on which MPSC tuples were covered by which test cases. We were not

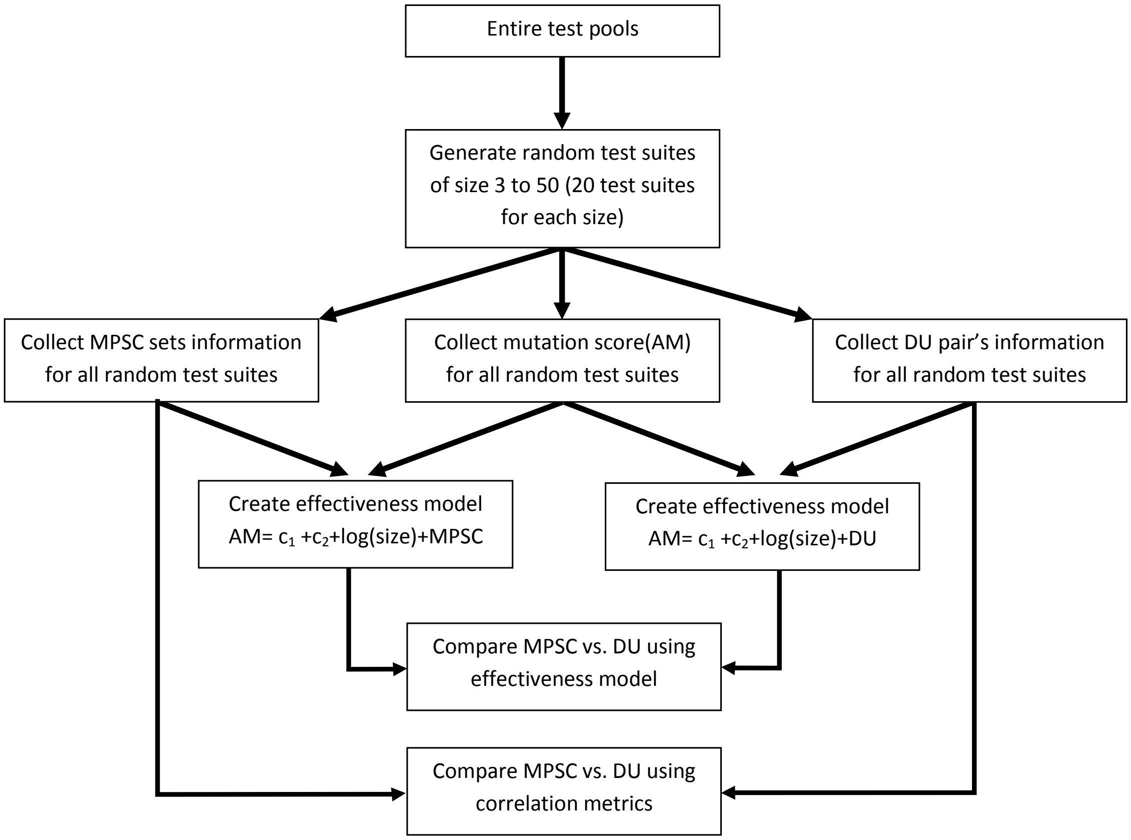

Entire test pools
Generate random test suites of size 3 to 50 (20 test suites for each size)
Collect MPSC sets information for all random test suites
Collect mutation score(AM) for all random test suites
Collect DU pair's information for all random test suites
Create effectiveness model AM= c1 +c2+log(size)+MPSC
Create effectiveness model AM= c1 +c2+log(size)+DU
Compare MPSC vs. DU using effectiveness model
Compare MPSC vs. DU using correlation metrics

Figure 5.1: MPSC vs. DU analysis process

able to run 20 of the test cases for jtopas, due to the problem with arrays and DUAF noted in section 3.5.

Using Andrews' tool mutgen [6], we generated a full set of mutant descriptions for each of the source programs. We then randomly selected mutant descriptions, generated the mutant program, compiled it, and ran the full suite of test cases on the mutated program until we obtained 100 non-equivalent mutants for each subject program [1]. We recorded which test cases killed which mutants. A detailed description of the mutant collection process was given in section 3.8.3.

Finally, we generated 20 randomly-chosen test suites for each size of test suite from 3 to 50 test cases, for a total of 960 test cases per subject program. Using the information about which test cases killed which mutants, we computed the effectiveness of each test suite as the percentage of mutants killed by the test suite. We also computed the cumulative MPSC or (for each collected g and p) and def-use coverage of each test suite based on the sets of MPSC tuples and def-use pairs covered by each test case. In Figure 5.1 we show the analysis process in a flow graph.

5.2 Relationship Between the Criteria

In Figure 5.2 we show a scatter plot of def-use vs MPSC for the subject program xml-security, where gap size $g = 0$ and number of points $p = 2$ (equivalent to branch coverage)[2]; each point in the plot represents one test case. Other programs showed the same strongly linear relationship.

We measured the Pearson correlation of MPSC coverage and def-use coverage for all programs, finding that it was greater than 0.87 for all programs. Correlation seems to be better for

[1] An equivalent mutant is one which is not killed (detected) by any test case. Equivalence of mutants is undecidable in theory and often difficult in practice. Here, we follow standard practice and approximate by considering every mutant not killed by any test case in the test pool as equivalent.

[2] In all scatter plots, the straight line shows the best linear fit, and the curved line shows the LOWESS (locally-weighted best fit) smoothing line. We use number of def-use pairs and/or MPSC tuples covered, rather than percentages, because we are interested in relationships which do not depend on the scales of the axes.

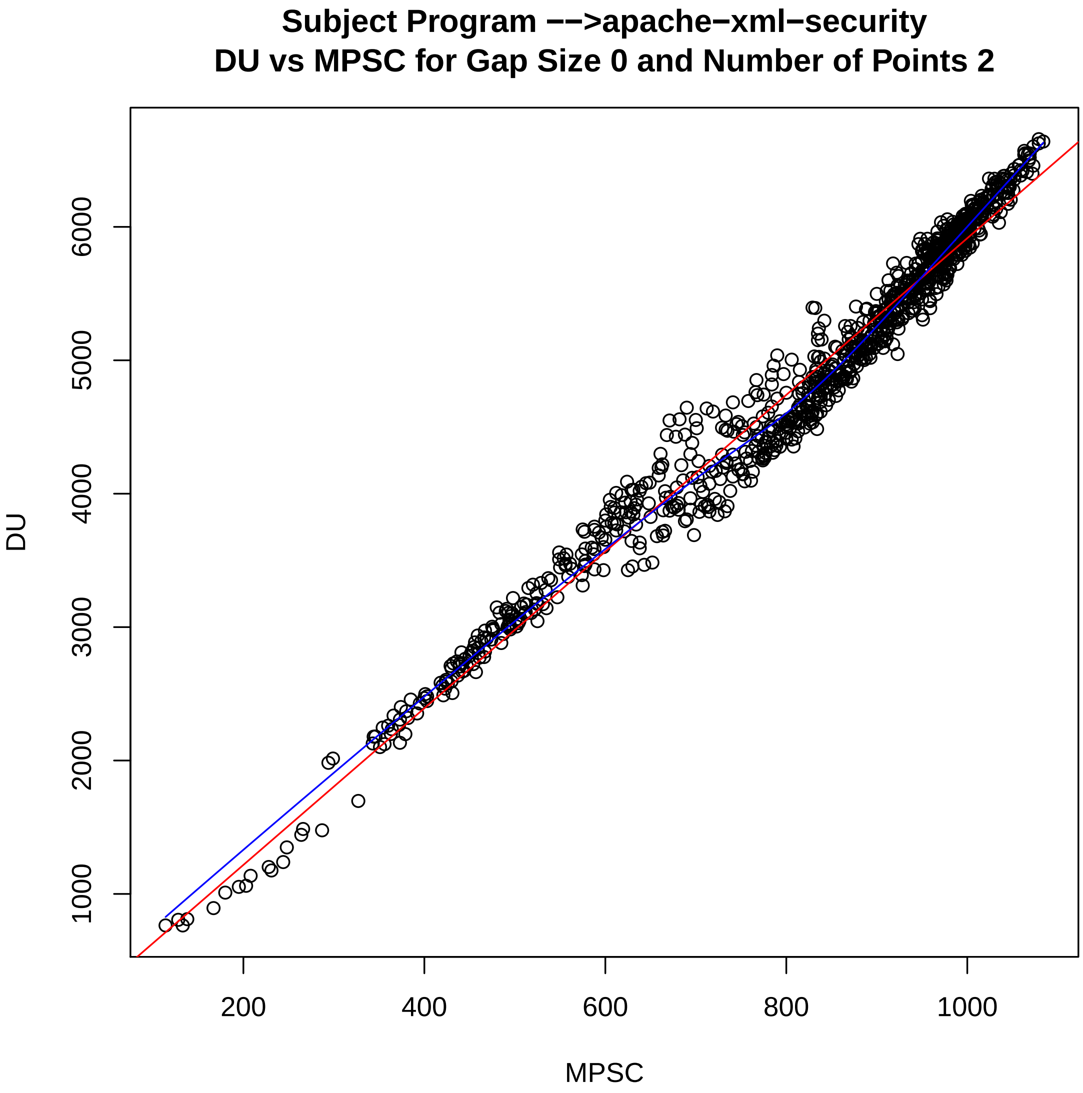


Figure 5.2: MPSC vs. DU for $g = 0$ and $p = 2$, subject program `Xml-security`.

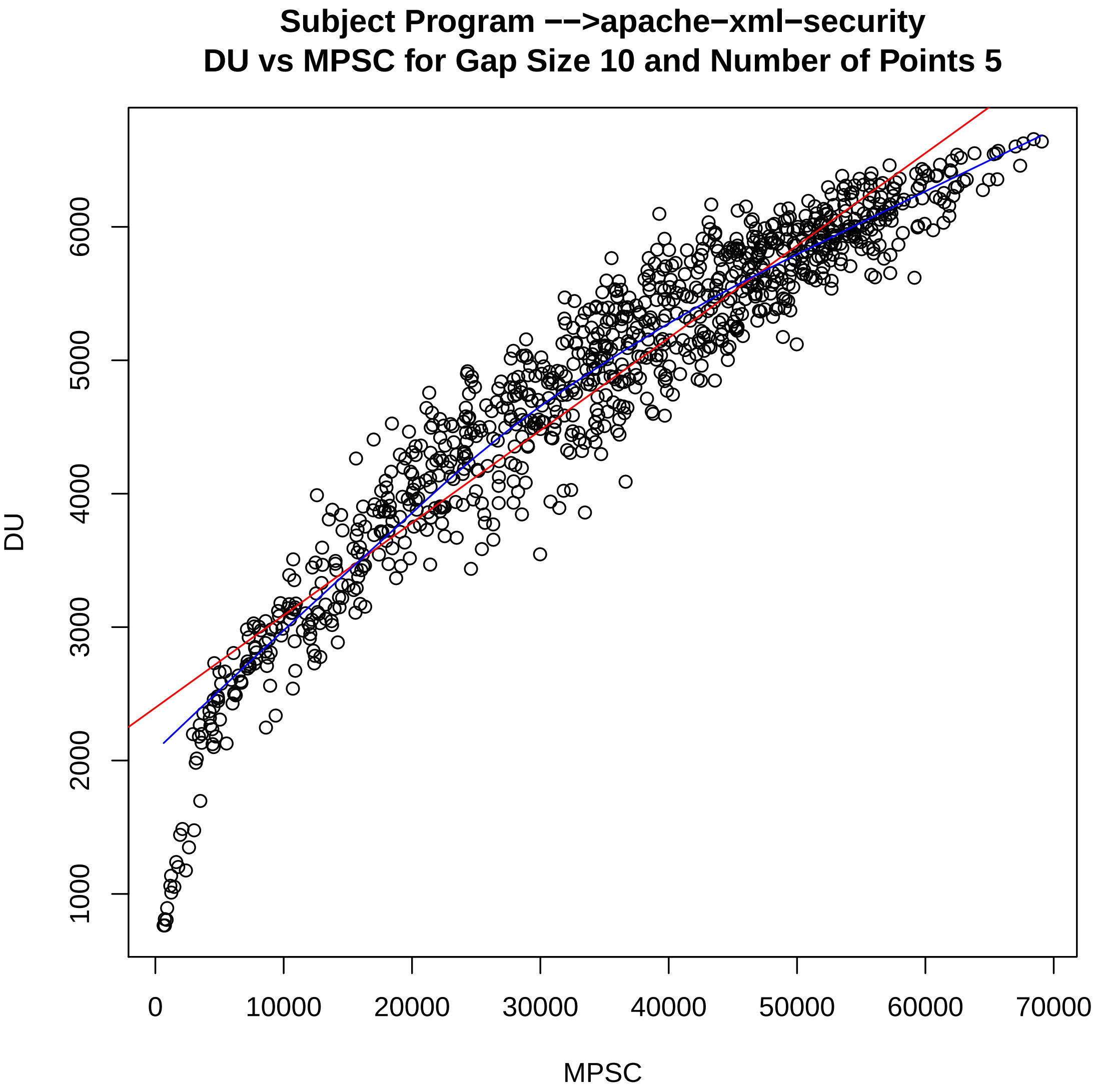


Figure 5.3: MPSC vs. DU for g = 10 and p = 5, subject program `Xml-security`.

larger programs. We also specifically analyzed the relationship between branch and DU coverage. Table 5.1 shows the result. Here, column 3 represents the correlation between Branch and DU, Column 4 represents the values of g and p that best correlate to DU, and the values of those correlations in column 5. This result indicates that even simple branch coverage is very strongly correlated with def-use coverage. We discuss the implications of this in chapter 7. Higher correlation of MPSC and DU suggests that the MPSC of a subject program is a good predictor of the DU of that subject program.

Spearman correlation (see Table 5.2) suggests that if MPSC of a test suite is less than another suite then it is more likely that DU of that suite will also be smaller. Finally in Table 5.3 we also present Kendall correlation measure; it implies MPSC and DU are not independent with each other.

We also analyzed the relationship between MPSC and other variations of data flow coverage like *C-use* and *P-Use*[3]. In Table 5.4 we present the correlation data. It is obvious from the table that P-use has a stronger relationship with MPSC than to C-use. In the last column we indicate the best data flow coverage based on correlation metrics: in most programs DU is best related to MPSC, but for some programs P-use narrowly exceeds DU.

As we increase the value of g and p, the linear relationship between MPSC and def-use coverage deteriorates. Figure 5.3 shows a similar scatter plot to Figure 5.2, but for $g = 10$ and $p = 5$. We note that MPSC distinguishes between test suites (assigns different coverage values) more than def-use does at these levels.

5.3 Predicting Effectiveness

We first visualized the effectiveness of the 960 test suites that we ran for each program. Figure 5.4 shows a typical such visualization: each point is one test suite, the X axis shows the cumulative number of MPSC tuples covered for $g = 9$ and $p = 2$, and the Y axis shows the

[3] Unlike ATAC, DuaF doesn't show *C-use* and *P-Use* coverage information but only provides Def-use coverage. Therefore, we weren't able to check the Pearson correlation for *C-use* and *P-Use* for Java programs.

Table 5.1: Pearson correlation for DU vs. Branch, and DU vs. MPSC.

	Program	Branch vs. DU	Best (g,p)	MPSC vs. DU
1	`flex`	0.9875	(1,2)	0.9925
2	`grep`	0.9949	(1,2)	0.9959
3	`printtokens`	0.9676	(0,2)	0.9676
4	`printtokens2`	0.9874	(0,2)	0.9874
5	`replace`	0.9810	(0,2)	0.9810
6	`schedule`	0.8718	(0,2)	0.8718
7	`schedule2`	0.9293	(0,2)	0.9293
8	`tcas`	0.8902	(5,5)	0.9056
9	`totinfo`	0.9750	(0,2)	0.9750
10	`jtopas`	0.9322	(1,4)	0.9464
11	`nanoxml`	0.9736	(1,3)	0.9877
12	`xml-security`	0.9888	(2,2)	0.9936

Table 5.2: Spearman correlation for DU vs. Branch, and DU vs. MPSC.

	Program	Branch vs. DU	Best (g,p)	MPSC vs. DU
1	`flex`	0.9882	(1,2)	0.9914
2	`grep`	0.99	(1,2)	0.9924
3	`printtokens`	0.9546	(0,2)	0.9546
4	`printtokens2`	0.8837	(6,2)	0.9141
5	`replace`	0.9457	(2,2)	0.9516
6	`schedule`	0.8433	(0,2)	0.8433
7	`schedule2`	0.9089	(0,2)	0.9089
8	`tcas`	0.8182	(4,3)	0.8378
9	`totinfo`	0.8991	(0,2)	0.8991
10	`jtopas`	0.9262	(1,3)	0.9329
11	`nanoxml`	0.9714	(1,3)	0.9868
12	`xml-security`	0.9892	(1,2)	0.9924

Table 5.3: Kendall correlation for DU vs. Branch, and DU vs. MPSC.

	Program	Branch vs. DU	Best (g,p)	MPSC vs. DU
1	`flex`	0.9158	(1,2)	0.9270
2	`grep`	0.9194	(1,2)	0.9304
3	`printtokens`	0.8529	(0,2)	0.8529
4	`printtokens2`	0.7733	(0,2)	0.7733
5	`replace`	0.828	(0,2)	0.828
6	`schedule`	0.7511	(0,2)	0.7511
7	`schedule2`	0.8204	(0,2)	0.8204
8	`tcas`	0.7387	(5,2)	0.7478
9	`totinfo`	0.8519	(0,2)	0.8519
10	`jtopas`	0.7772	(1,3)	0.7878
11	`nanoxml`	0.8624	(1,3)	0.9060
12	`xml-security`	0.9162	(1,2)	0.9293

Table 5.4: Pearson correlation for C-use vs. Branch, and C-use vs. MPSC. DF: Data-Flow coverages (DU, P-use, C-Use)

	Program	Branch vs. C-use	Best (g,p)	MPSC vs. C-use	Branch vs. P-use	Best (g,p)	MPSC vs. P-use	Best DF vs. MPSC
1	`flex`	0.9776	(4,2)	0.9832	0.9890	(1,2)	0.9938	P-use
2	`grep`	0.9911	(1,2)	0.9916	0.9945	(1,2)	0.9957	DU
3	`printtokens`	0.8597	(0,2)	0.8597	0.9793	(3,2)	0.9817	P-use
4	`printtokens2`	0.9508	(0,2)	0.9508	0.9864	(0,2)	0.9864	DU
5	`replace`	0.9394	(0,2)	0.9394	0.9843	(0,2)	0.9843	P-use
6	`schedule`	0.4715	(1,3)	0.4847	0.8797	(0,2)	0.8797	P-use
7	`schedule2`	0.5389	(2,2)	0.5655	0.9258	(0,2)	0.9258	DU
8	`tcas`	0.8435	(5,2)	0.8438	0.8791	(5,5)	0.8999	DU
9	`totinfo`	0.8849	(0,2)	0.8849	0.9645	(0,2)	0.9645	DU

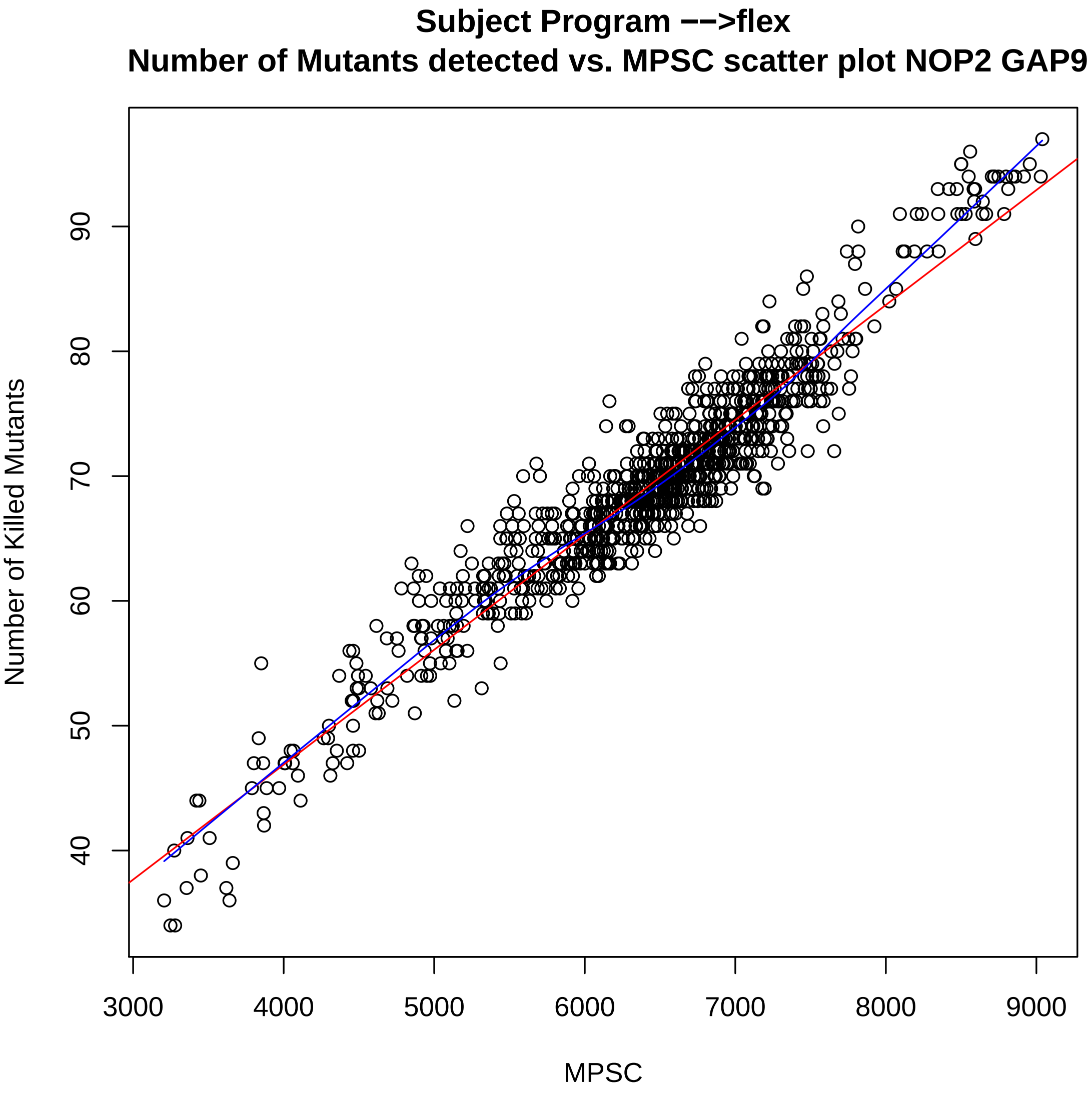


Figure 5.4: MPSC vs. number of mutants detected for $g = 9$ and $p = 2$, subject program `flex`.

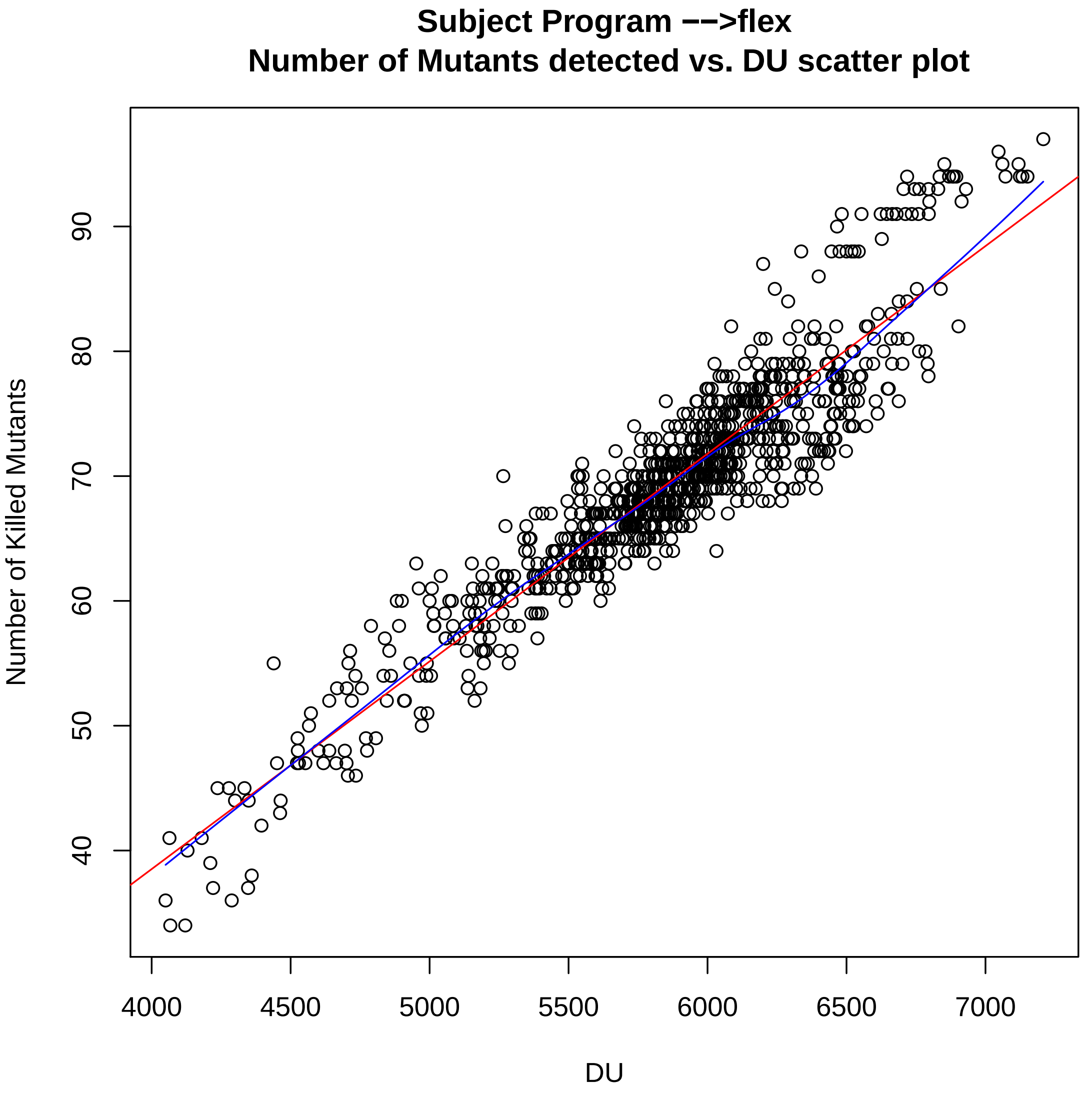


Figure 5.5: DU vs. number of mutants detected, subject program `flex`.

number of mutants killed. Similar graphs were also produced for DU (see Figure 5.5). In both of the graphs, effectiveness increases with coverage, and for many of the subject programs the relationship follows a linear pattern. So we applied a simple linear regression model as follows -

$$AM = c_1 + c_2 \cdot coverage \tag{5.1}$$

where *AM* (Adequacy on Mutants), is the percentage of mutants killed by the test suite, and *coverage* is the coverage of the test suite, in MPSC tuples or def-use pairs. We did this for each value of g and p and for def-use. We measured the accuracy of the resulting models by the adjusted R^2 measure. The results are summarized in Table 5.5. Column 4 gives the setting of g and p that resulted in the most accurate linear model for MPSC. Columns 3, 5 and 6 show the adjusted R^2 value of the models from branch coverage ($g = 0, p = 2$), from the best setting, and from def-use coverage. The boldface number in each row is the adjusted R^2 of the most accurate model.

For 11 of the 15 subjects, the accuracy of all models ranked as “high” or “very high” (> 0.70) on the standard Guilford scale [42]. For two subjects, the accuracy of all models ranked as “low” or “moderate” (between 0.40 and 0.70) and for the last two subjects, the accuracy of all models ranked as “moderate” or “high”.

When a test case is added to a test suite, it cannot *decrease* the suite’s effectiveness; on average, it increases it by a given nonzero amount. In equation 5.1 we did not consider size of a test suite, so the effectiveness model is not complete.

Siami Namin and Andrews [87] found that both size and coverage contributed to an accurate prediction of test suite effectiveness. We therefore refine RQ3 to the following: “Does using the size of a test suite and MPSC lead to a more accurate model of test suite effectiveness than using the size of the test suite and def-use coverage?” This formulation of the question accounts for the confounding factor of size in test suite effectiveness.

In Figure 5.6 we try to visualize the relationship between test suite size and number of

Table 5.5: Accuracy of effectiveness models according to Equation 5.1. “n/a”: not available due to ATAC limitations.

	Program	Branch Adj R^2	Best (g,p)	Best Adj R^2	DU Adj R^2
1	concordance	0.8831	(1,2)	**0.9005**	n/a
2	flex	0.8291	(7,2)	**0.9133**	0.868
3	gzip	0.7977	(1,2)	**0.806**	n/a
4	grep	0.8726	(3,2)	**0.8885**	0.8834
5	sed	**0.9523**	(0,2)	0.9523	n/a
6	printtokens	**0.9462**	(0,2)	0.9462	0.845
7	printtokens2	0.8926	(0,2)	0.8926	**0.9013**
8	replace	0.8216	(3,2)	**0.8341**	0.8165
9	schedule	0.5751	(1,2)	**0.6095**	0.5368
10	schedule2	0.1986	(1,5)	**0.4263**	0.3042
11	tcas	0.6903	(3,5)	**0.7532**	0.7447
12	totinfo	0.7446	(0,2)	0.7446	**0.7521**
13	jtopas	0.6889	(1,2)	**0.7036**	0.6332
14	nanoxml	0.8934	(4,2)	**0.913**	0.888
15	xml-security	0.8827	(2,2)	0.8834	**0.8867**

mutants killed using a scatter plot; we also show the scatter plot for size vs. DU and MPSC in Figures 5.7 and 5.8 respectively. No direct linear relationship seems to exist between them. However, a complex, nonlinear relationship exists between the measures of test suite size, test suite coverage and test suite effectiveness. The nonlinearity is clear from the fact that the points form a curve rather than a line in those graphs. These relationships are similar to that reported by Siami Namin and Andrews [87]. There is no obvious difference between MPSC and DU that can be deduced by just looking at the scatter plots.

The refined RQ3 essentially factors out this coverage-neutral amount of added effectiveness, allowing us to measure how much additional value we get out of increases in coverage. To answer the refined question, we used R to fit models of the form [4]

$$AM = c_1 + c_2 \cdot log(size) + c_3 \cdot coverage \tag{5.2}$$

[4] Siami Namin and Andrews [87] proposed that model.

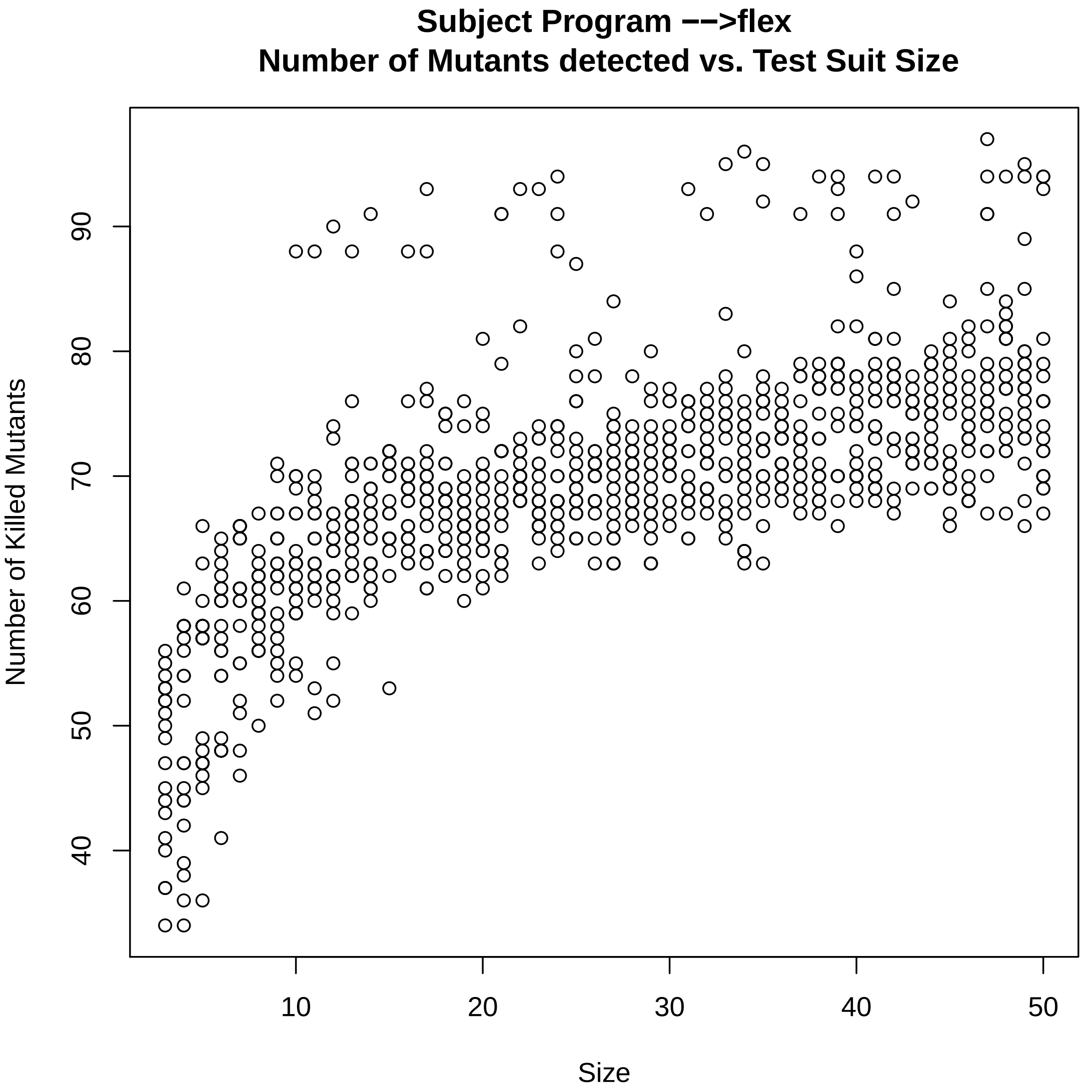


Figure 5.6: Test suite size vs. number of mutants detected, subject program `flex`.

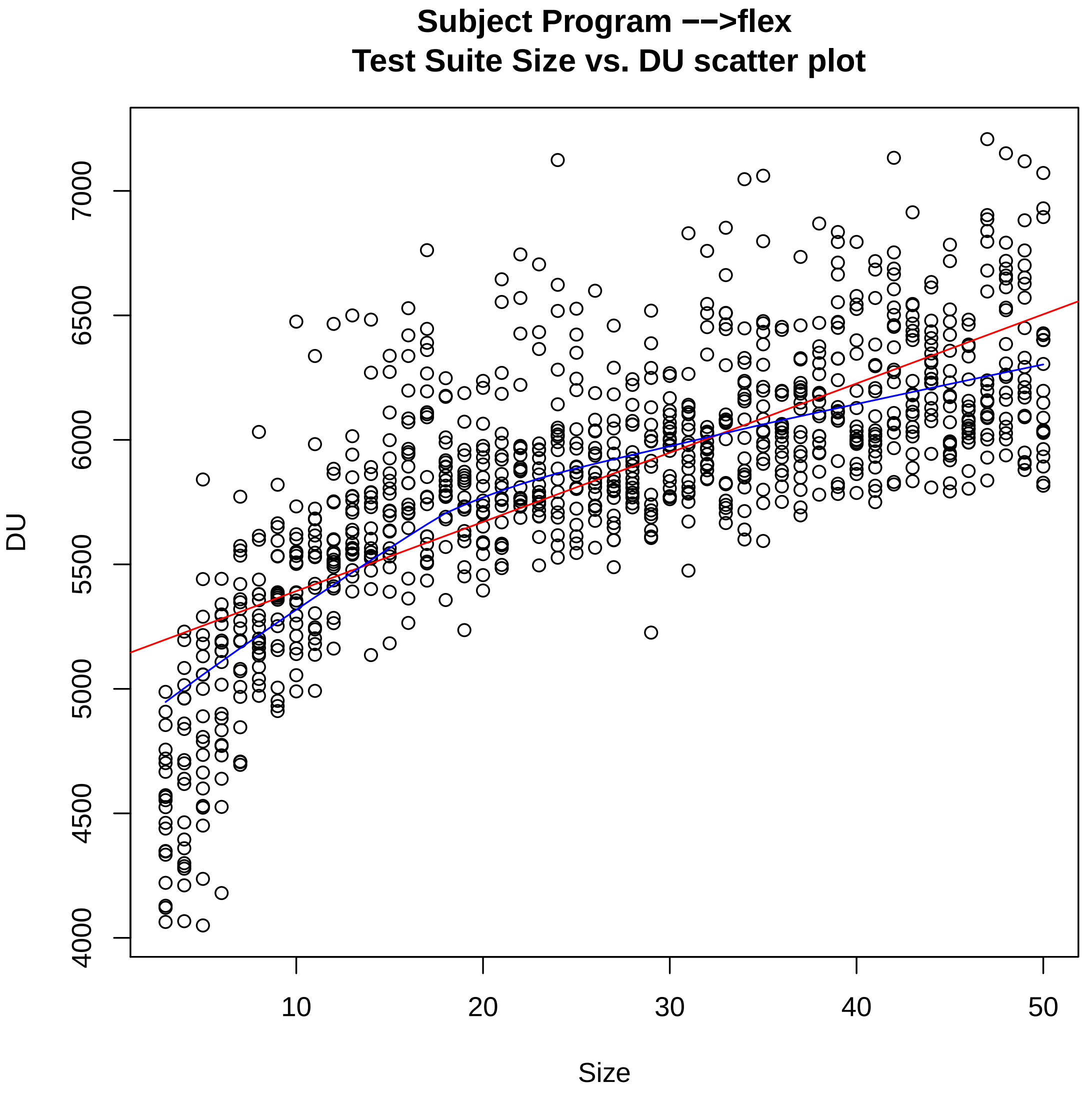


Figure 5.7: Test suite size vs. number of DU pairs, subject program `flex`.

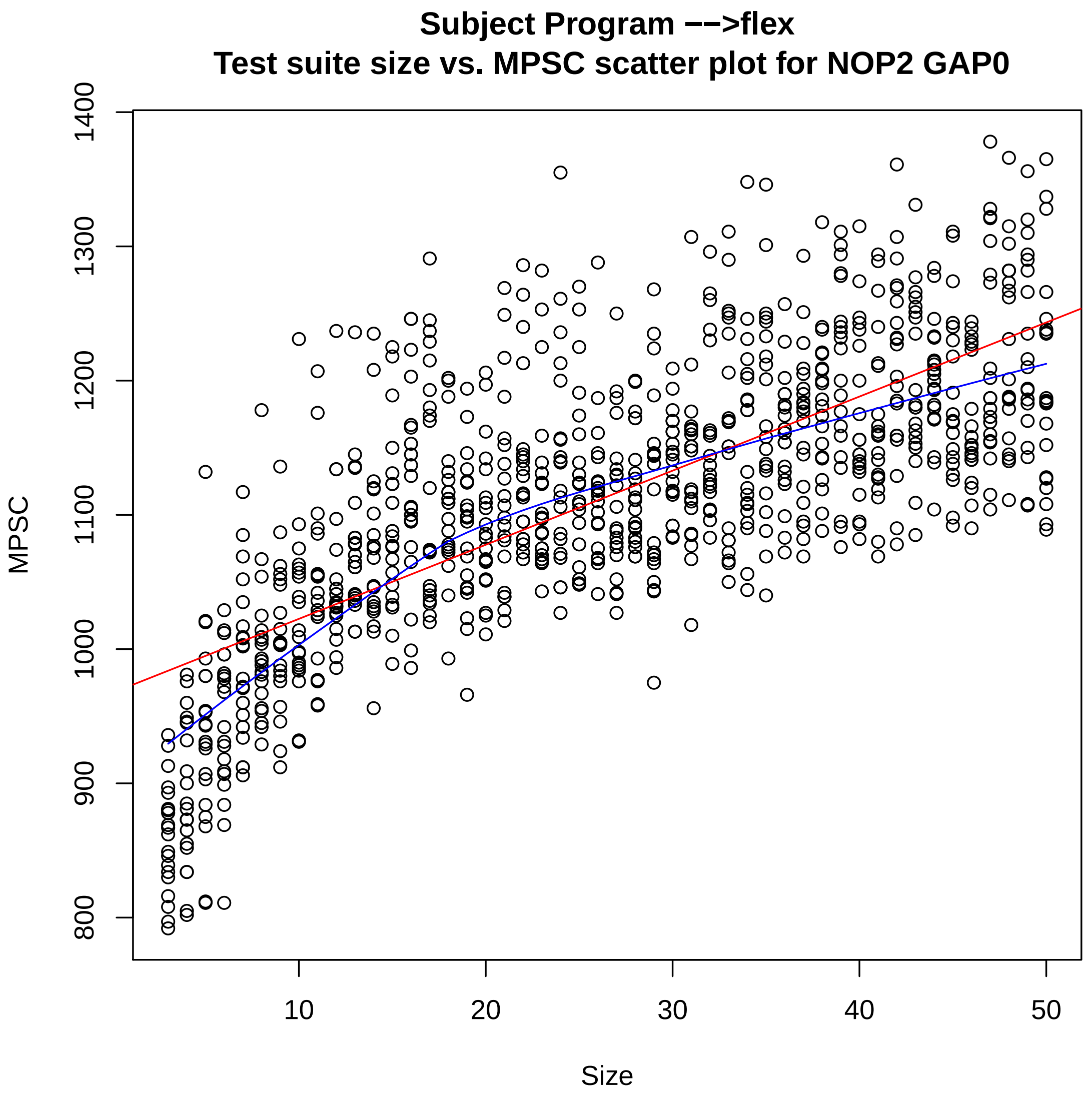


Figure 5.8: Test suite size vs. number of MPSC, subject program `flex`.

to our test suite data, where *size* is the number of test cases. Like the previous model (shown in Equation 5.1) we did this for each value of g and p and for def-use. The results are summarized in Table 5.6. We consider only cases in which both factors had a statistically significant effect on effectiveness ($p < 0.05$). For all programs the second model has better adjusted R^2 values when compared to the first model. For `gzip` and `totinfo`, in the branch coverage case, $log(size)$ did not have a statistically significant effect on effectiveness. For `gzip`, `totinfo` and `grep` among all the MPSC combinations where size has a significant effect, we can not get one that is better than the coverage-only model (that is, Equation 5.1). For `totinfo` size has no significant effects in coverage & size model (that is, equation 5.2) for DU coverage.

For 7 out of the 15 subjects, the most accurate model was the one yielded by branch coverage; for 7 others, it was the one yielded by some other setting of MPSC, and for the remaining 1 it was the one yielded by def-use coverage. However, for 8 out of the 12 (DU instrumented) subject programs (including the one for which def-use was the most accurate), the accuracy of the models resulting from branch coverage, from the best setting of MPSC, and from DU coverage were all within 0.05 of each other, indicating that there was little *practical* difference among the criteria (see the Figures 5.9 and 5.10 for the subject program `flex`). Figure 5.11 shows the model resulting from MPSC ($g = 9$, $p = 2$), which represents the best prediction model for subject program `flex`.

For 11 of the 15 subjects, the accuracy of all models ranked as “high” or “very high” (> 0.70) on the standard Guilford scale [42]. For the other four subjects, the accuracy of all models ranked as “low” or “moderate” (between 0.40 and 0.70). In Figure 5.12 and 5.13 we show the actual AM vs. predicted AM for `schedule2` using size and MPSC or DU respectively; they suggest that the model is good for neither DU nor MPSC. This indicates that for some subjects, there were confounding factors other than test suite size and coverage that affected test suite effectiveness.

It is also interesting to observe the patterns of prediction; they look very similar for both DU and MPSC in Figure 5.12 and 5.13 respectively even when the model does not fit well for

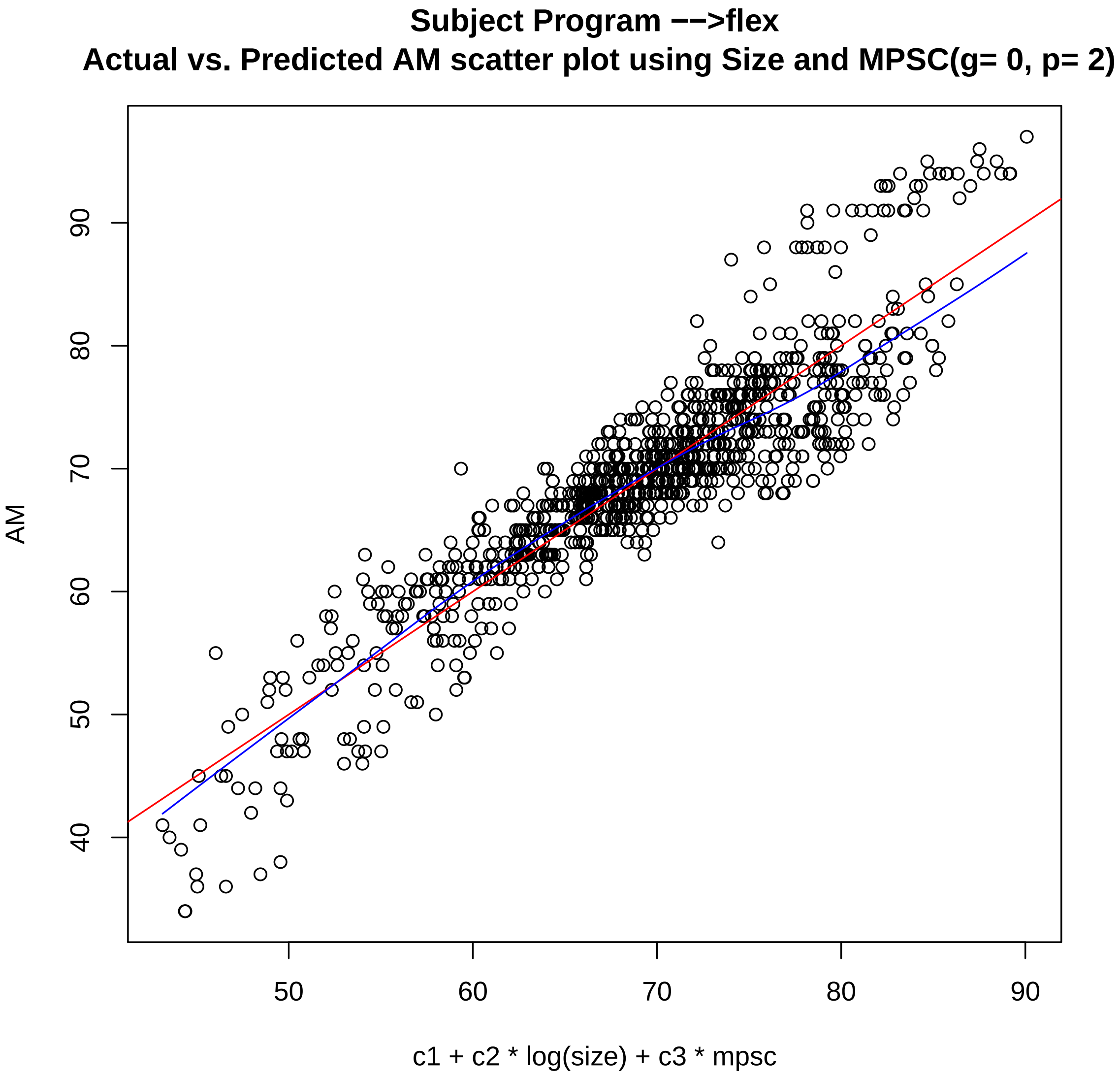


Figure 5.9: Actual AM vs. predicted AM using MPSC and size for $g = 0$ and $p = 2$ (that is, branch coverage), subject program `flex`.

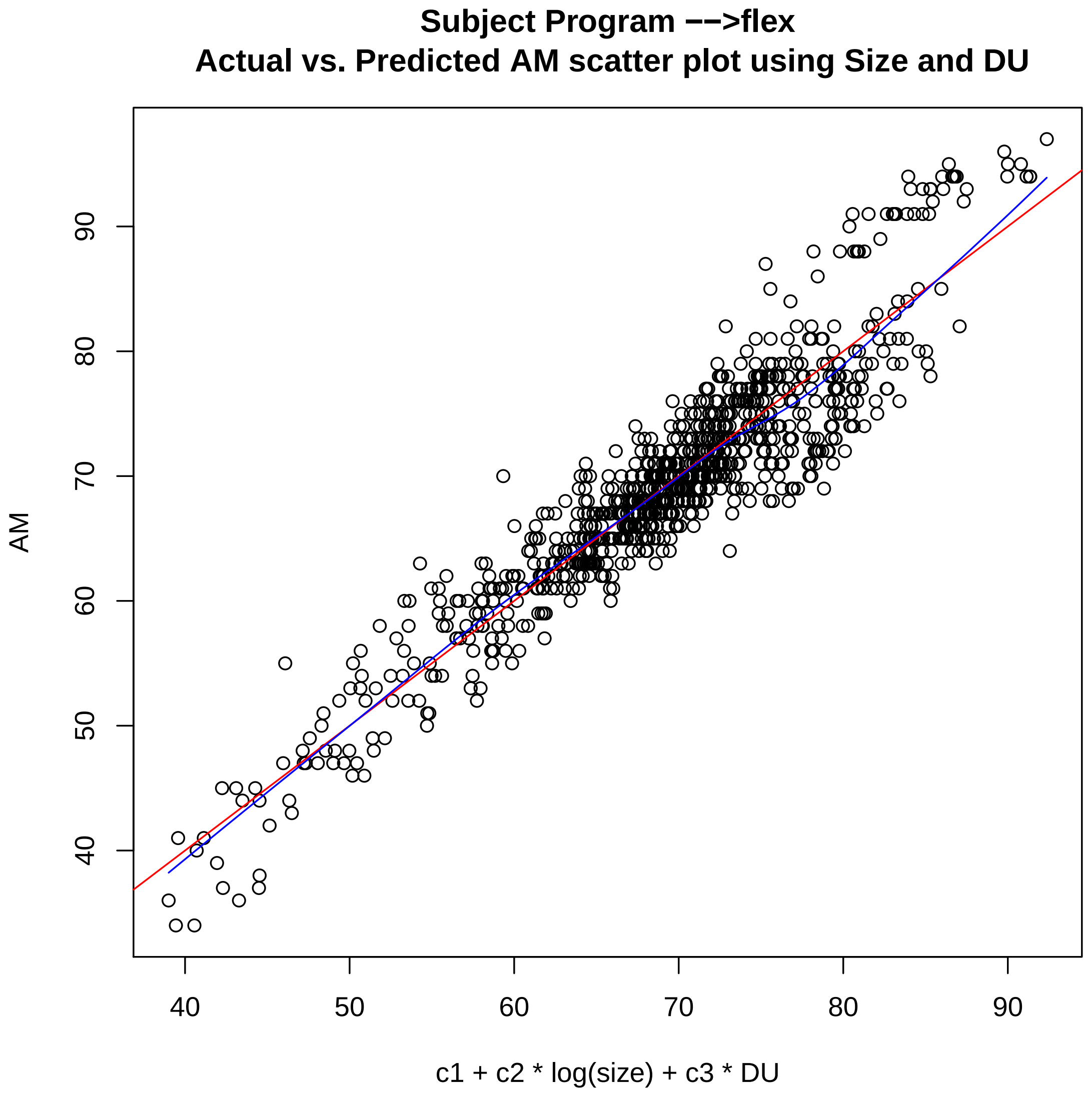


Figure 5.10: Actual AM vs. predicted AM using DU and size, subject program `flex`.

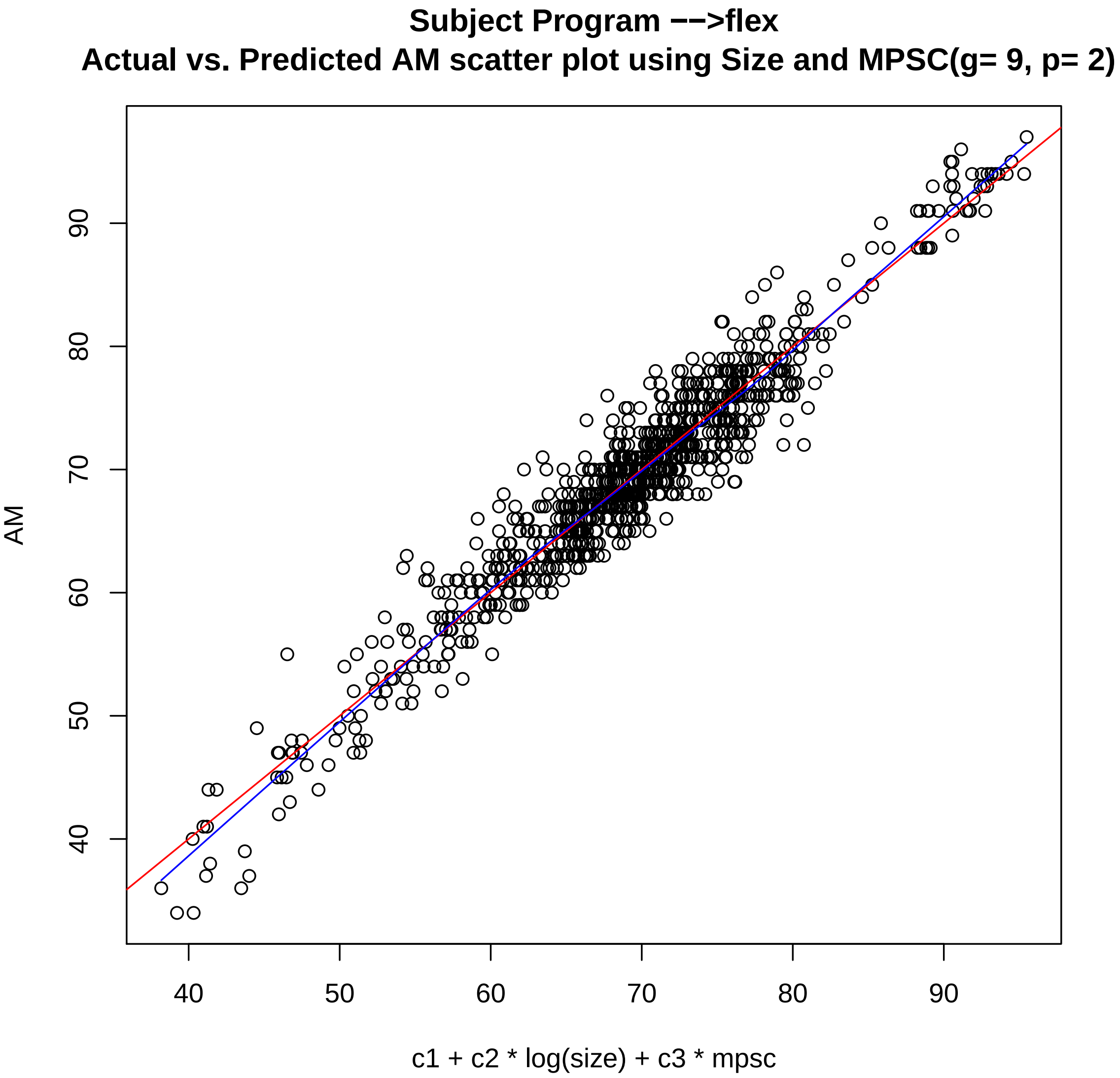


Figure 5.11: Actual AM vs. predicted AM using MPSC and size for $g = 9$ and $p = 2$, subject program `flex`.

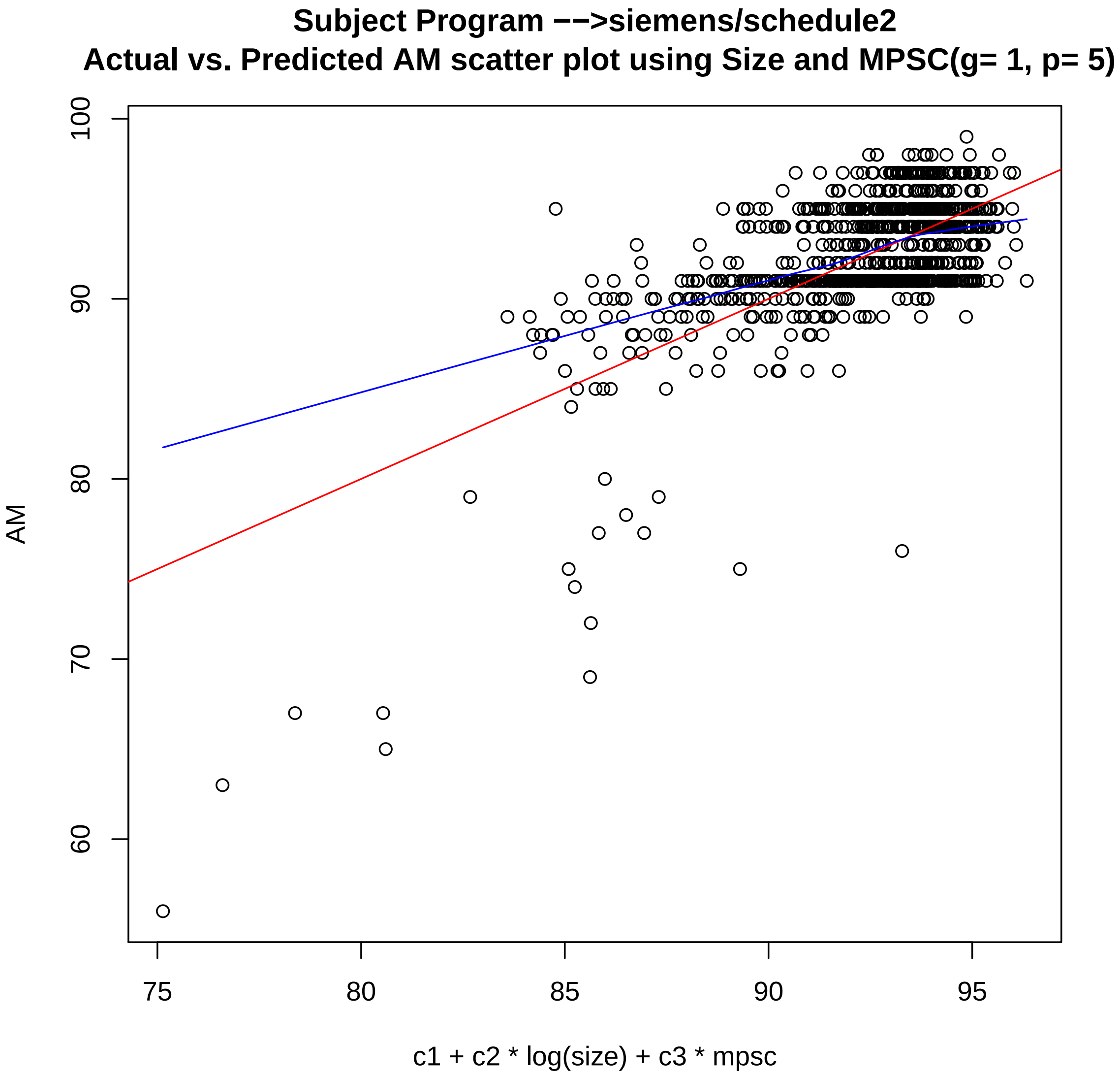


Figure 5.12: Actual AM vs. predicted AM using MPSC and size for $g = 1$ and $p = 5$, subject program `schedule2`.

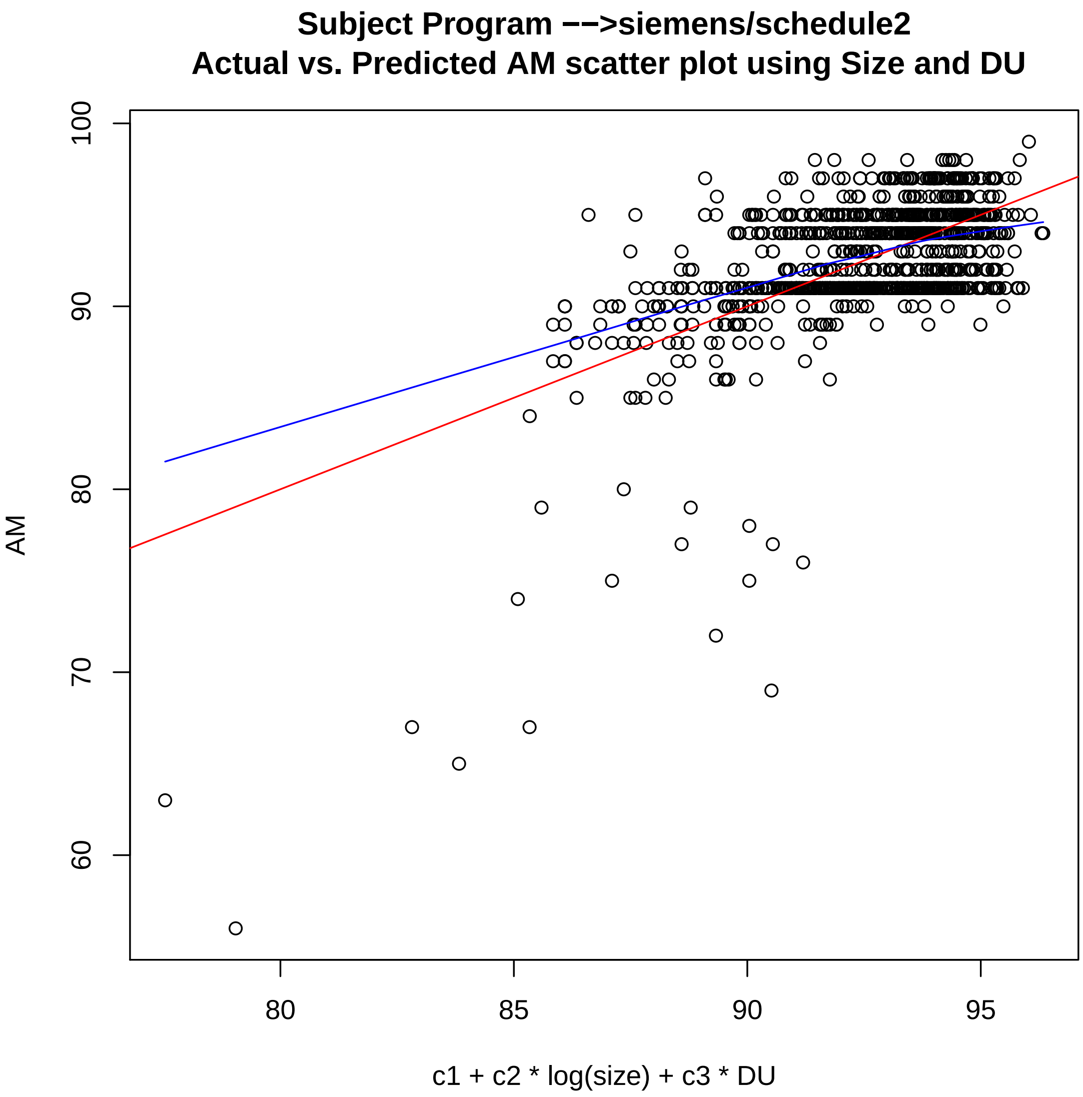


Figure 5.13: Actual AM vs. predicted AM using DU and size, subject program `schedule2`.

Table 5.6: Accuracy of effectiveness model according to Equation 5.2. “n/s”: not statistically significant. “n/a”: not available due to ATAC limitations.

		Branch	Best	Best	DU
	Program	Adj R^2	(g,p)	Adj R^2	Adj R^2
1	`concordance`	**0.9135**	(0,2)	0.9135	n/a
2	`flex`	0.8306	(9,2)	**0.9246**	0.8687
3	`gzip`	n/s	(4,3)	**0.6836**	n/a
4	`grep`	0.8759	(3,2)	**0.8890**	0.8853
5	`sed`	**0.9525**	(0,2)	0.9525	n/a
6	`printtokens`	**0.9491**	(0,2)	0.9491	0.8768
7	`printtokens2`	0.9041	(0,2)	0.9041	**0.9042**
8	`replace`	**0.8649**	(0,2)	0.8649	0.8641
9	`schedule`	**0.6095**	(0,2)	0.6095	0.5750
10	`schedule2`	0.3208	(1,5)	**0.4400**	0.3600
11	`tcas`	0.8557	(5,5)	**0.8678**	0.8314
12	`totinfo`	n/s	(1,2)	**0.6971**	n/s
13	`jtopas`	**0.7568**	(0,2)	0.7568	0.7515
14	`nanoxml`	0.9014	(1,5)	**0.9066**	0.888
15	`xml-security`	**0.888**	(0,2)	0.888	0.8869

both of them. This also supports our observation that DU and MPSC are highly correlated.

5.4 Summary

- A high degree of correlation exists between MPSC and data flow coverage.
- Compared to C-use, P-use is highly correlated with MPSC. In our experiment, some cases showed a slightly higher correlation with MPSC than DU.
- The linear relationship between def-use and MPSC is stronger for lower g and p values and weaker for higher values.
- MPSC and DU showed a similar trend in mutation score which suggests they are similar in terms of predicting the fault-detection capability of a test suite.
- Surprisingly, simple branch coverage is as good as DU or higher-order MPSC in predicting the effectiveness of a test suite for most programs.

Chapter 6

Implementation and Performance

The JQXZ utility instruments source code to make calls to a library which collects coverage information. The implementations of the library used for the experiments above simply recorded each branch executed in a file. We also developed two other prototype implementations for Java, a hash set implementation and a bitset implementation, which were intended to be closer to implementations that could be used in production environments.

In this section, we describe these implementations and give the results of performance experiments we ran to measure the overhead of MPSC collection and compare it to the overhead of def-use coverage.

6.1 Prototype Implementations

Both prototype implementations of the Java library depended on a hash function for MPSC tuples. We implemented several hash functions, but we found in exploratory performance experiments that the best used a simple algorithm of the same form as that of the `java.lang.String` hash function.

The implementations maintain a circular buffer representing the last $g(p-1)+1$ branches executed. The branches are identified by source file, line and character number, and (for decisions) whether true or false. Each branch causes an element to be added to the circular buffer and the newly completed MPSC tuple to be recorded as covered.

The hash set implementation stores each covered tuple in a hash set. The initial size of the hash set can be computed based on the maximum expected number of tuples needed, as outlined in Section 4.3.2. The advantage of this implementation is that the exact set of tuples is recorded. The disadvantage (compared to the bitset implementation) is that a larger amount of storage is needed.

The bitset implementation computes the hash code *h* of each covered tuple, and stores it by setting the *h*-th bit in a `java.util.BitSet` to 1. The advantage of this implementation over the hash set is that much less storage is needed. The disadvantages are that it is impossible to extract which tuples have been covered, and that two or more tuples may hash to the same value, losing information about precisely how many tuples have been covered. This implementation might still be useful, for instance for automated test input generation schemes which only have to compute whether one test case has covered some line of code not covered earlier.

6.2 Procedure

To measure the performance of our implementations and compare it to that of def-use, we chose our three Java subject programs and added two programs used by Santelices and Harrold [83], `tcas` (a Java translation of the Siemens program) and `scimark2` (a JVM performance benchmark). We ran the test suite for each program on the uninstrumented program, the program instrumented for def-use coverage by the state-of-the-art tool DUAF developed by Santelices [83], and the program instrumented for MPSC by our tool. We were not able to successfully instrument `scimark2` for DU coverage with the current version of DUAF.

In Figure 6.1 we show the performance-analysis process. Our two main concerns were - a performance comparison between MPSC and DU and the investigation of information loss due to the bitset implementation.

For our tool, we ran the program for every setting of g from 0 to 10 and p from 2 to

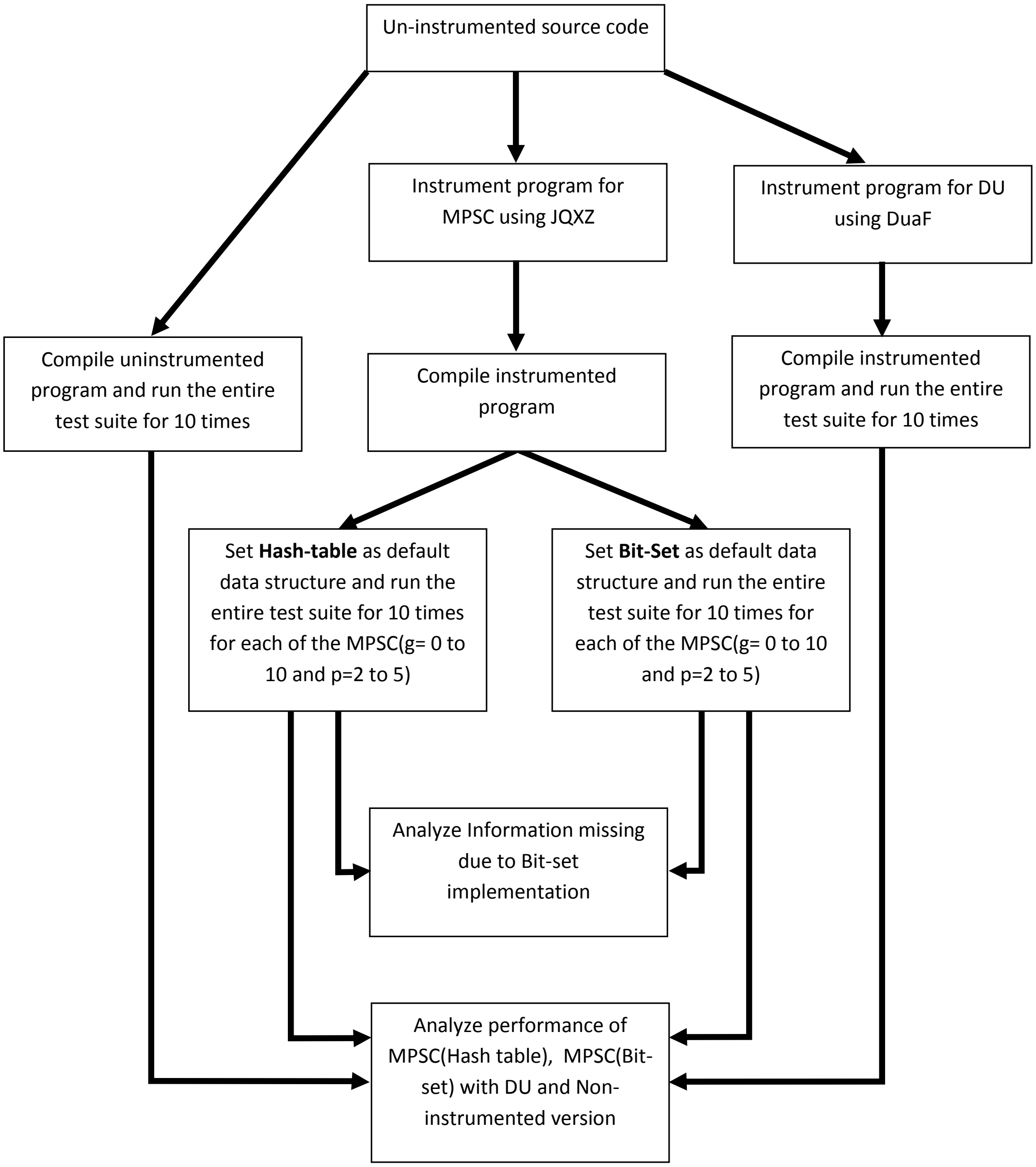

Un-instrumented source code
Instrument program for MPSC using JQXZ
Instrument program for DU using DuaF
Compile uninstrumented program and run the entire test suite for 10 times
Compile instrumented program
Compile instrumented program and run the entire test suite for 10 times
Set **Hash-table** as default data structure and run the entire test suite for 10 times for each of the MPSC(g= 0 to 10 and p=2 to 5)
Set **Bit-Set** as default data structure and run the entire test suite for 10 times for each of the MPSC(g= 0 to 10 and p=2 to 5)
Analyze Information missing due to Bit-set implementation
Analyze performance of MPSC(Hash table), MPSC(Bit-set) with DU and Non-instrumented version

Figure 6.1: Flow graph to show the performance-analysis process.

Table 6.1: Performance data. Uninst: uninstrumented

	Uninst	DU overhead (%)		MPSC overhead (%)	
Subject	(CPU sec)	SH	HA	Hashset	Bitset
`jtopas`	0.3117	–	2078	460.2	594.1
`nanoxml`	0.1862	–	71.60	97.90	70.15
`xml-sec`	4.487	–	86.90	36.69	31.37
`tcas`	0.1724	17.91	11.91	5.524	5.376
`scimark2`	29.31	160.4	–	0.4389	-2.703

5 for both hashtable and bitset implementation. We ran each program 10 times, except for `scimark2`, which we ran 100 times because it uses a random number generator to generate some of its test data. We also followed the same approach for the DU instrumented and non-instrumented versions. For the purposes of comparison, we used a coverage-reporting mechanism that was stylistically similar to that used by DuaF [1].

We measured CPU time by the "user" time reported by the Unix `time` facility in `bash`. We then calculated the average time to run one test case or (for `scimark2`) the average time to run the program as a whole. We averaged the results for all values of g and p to get a summary value for MPSC.

For information loss we collected reported tuples for both hashtable and bitset for the entire test suite and compared them. We will analyze the details in the following sections.

6.3 Performance

Any kind of instrumentation of a source code adds overhead to the original software: in general, processing time increases. Performance comparison shows the relative processing time increase due to instrumentation. In the previous section, we discussed the procedure of our performance experimentation for DU vs. MPSC. Table 6.1 shows the results of our experimen-

[1] After a successful run, DuaF reported all covered DU pairs on the standard output, a faster method than directly storing them to a disk file.

tation.

Results are reported for the uninstrumented program in CPU seconds, and for the other columns in percentage overhead (extra time needed) for the instrumentation. The experiments were run on a Sun UltraSPARC-IIIi with a 1.593GHz processor and 4GB of memory. For def-use coverage, we give the overhead reported in [83] where it is available (column SH), and also the overhead we calculated (column HA), since CPUs and JVMs can vary.

Table 6.1 shows that the overhead for our prototype bitset implementation was always less than that of DuaF, and that the overhead for our prototype hash set implementation was less than that of DUAF for every program except `nanoxml`. The overhead for DUAF was greater than both the average overhead for MPSC across all g and p, and the average overhead for any individual g and p, except in the case of `nanoxml` using the hash set implementation.

In the case of `scimark2`, paradoxically, not only did the MPSC instrumentation have very little effect, but on average the software instrumented with the bitset implementation took *less* time than the uninstrumented version. We attribute this to random noise resulting from the choice of random seeds by different runs, and possibly to operating system effects due to amount of memory allocated for a process.

The average run time for each subject program varied with different values of g and p, but there is no clear pattern across different subjects (see the Figure 6.2 and 6.3). In `Jtopas` (Figure 6.2) some kind of pattern may exist, such as an average run-time increase with higher p, but this is not true for `xml-sec` (Figure 6.3) or other subject programs. This was probably due to different rates of hash collisions for different MPSC.

Figures 6.4 shows sample run-time data for DU, MPSC, and non-instrumented versions of the subject program `Jtopas`. The test cases are arranged from left to right in ascending orders of DU coverage. Showing the data in ascending order highlights the difference between short-run test cases vs. long-run test cases. The graph shows that while neither DU nor MPSC incur much overhead for short-run test cases, DU performs substantially worse than MPSC in long-run test cases.

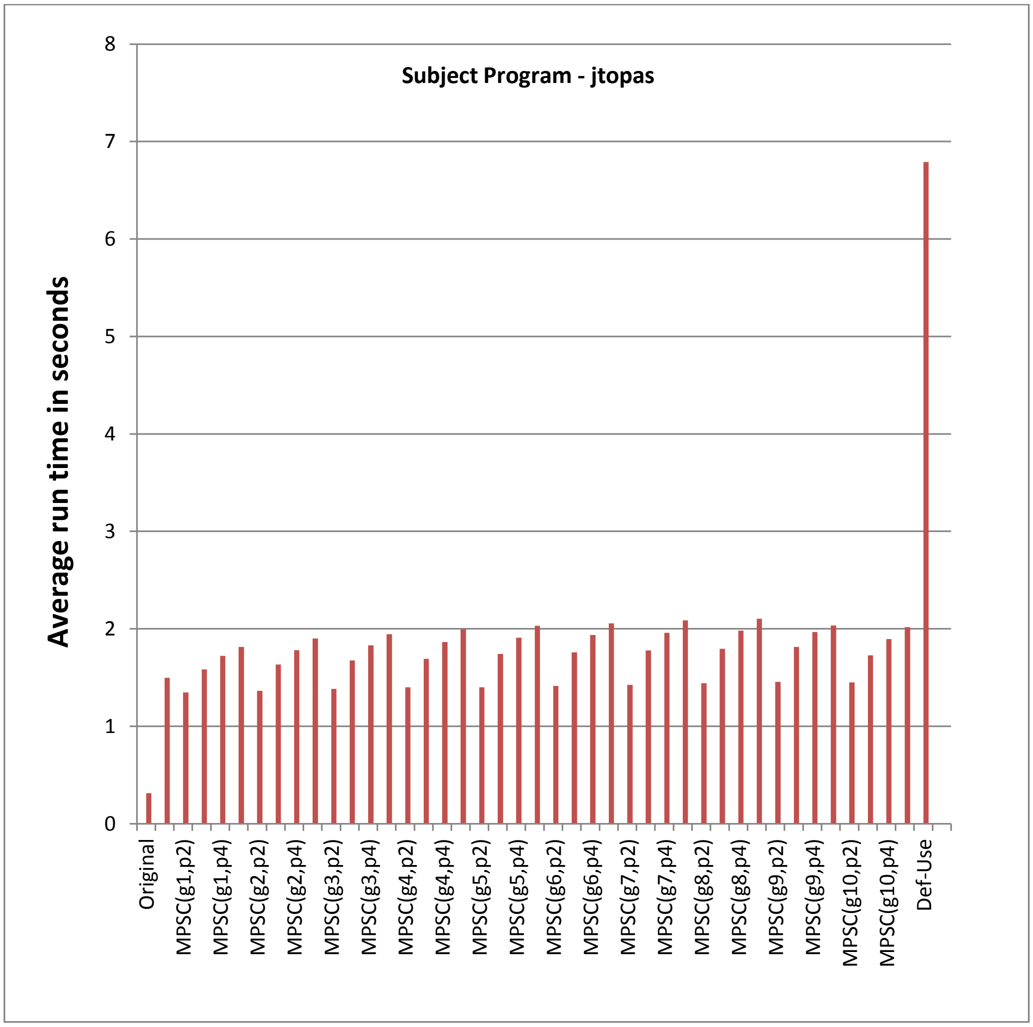


Figure 6.2: Average running time for instrumented (Def-use and MPSC) and non-instrumented version, subject program `Jtopas`.

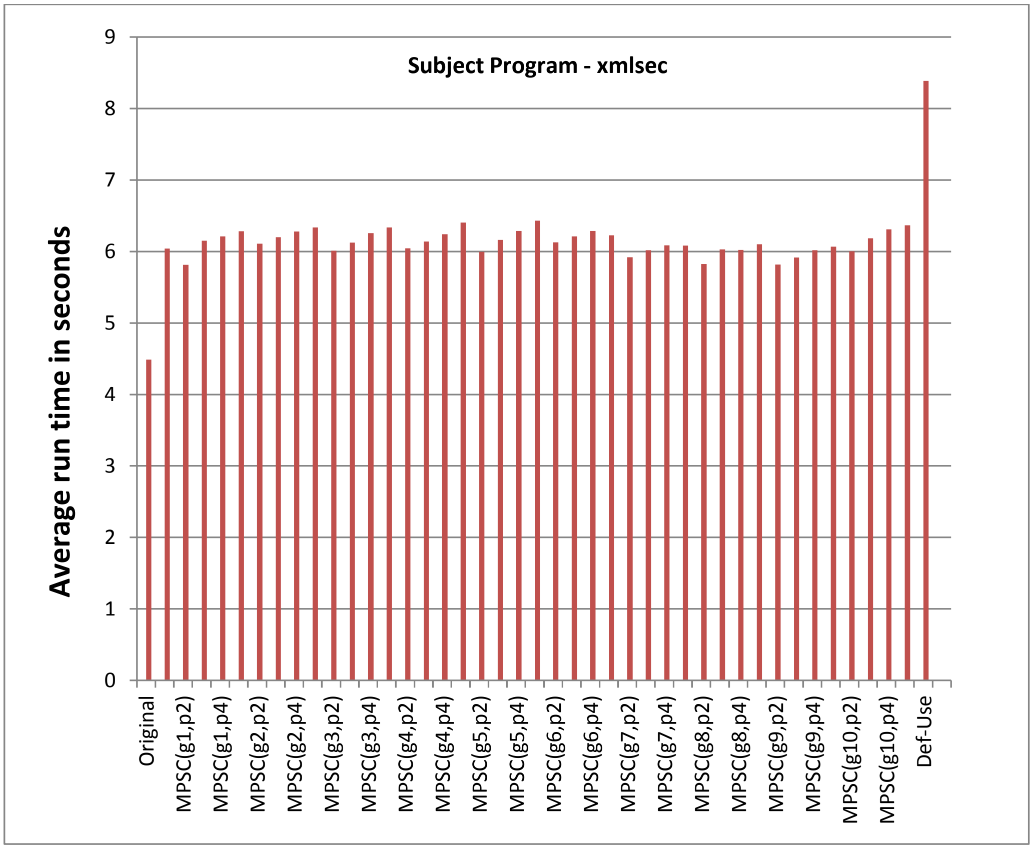


Figure 6.3: Average run time for instrumented (Def-use and MPSC) and non-instrumented version, subject program `xml-sec`.

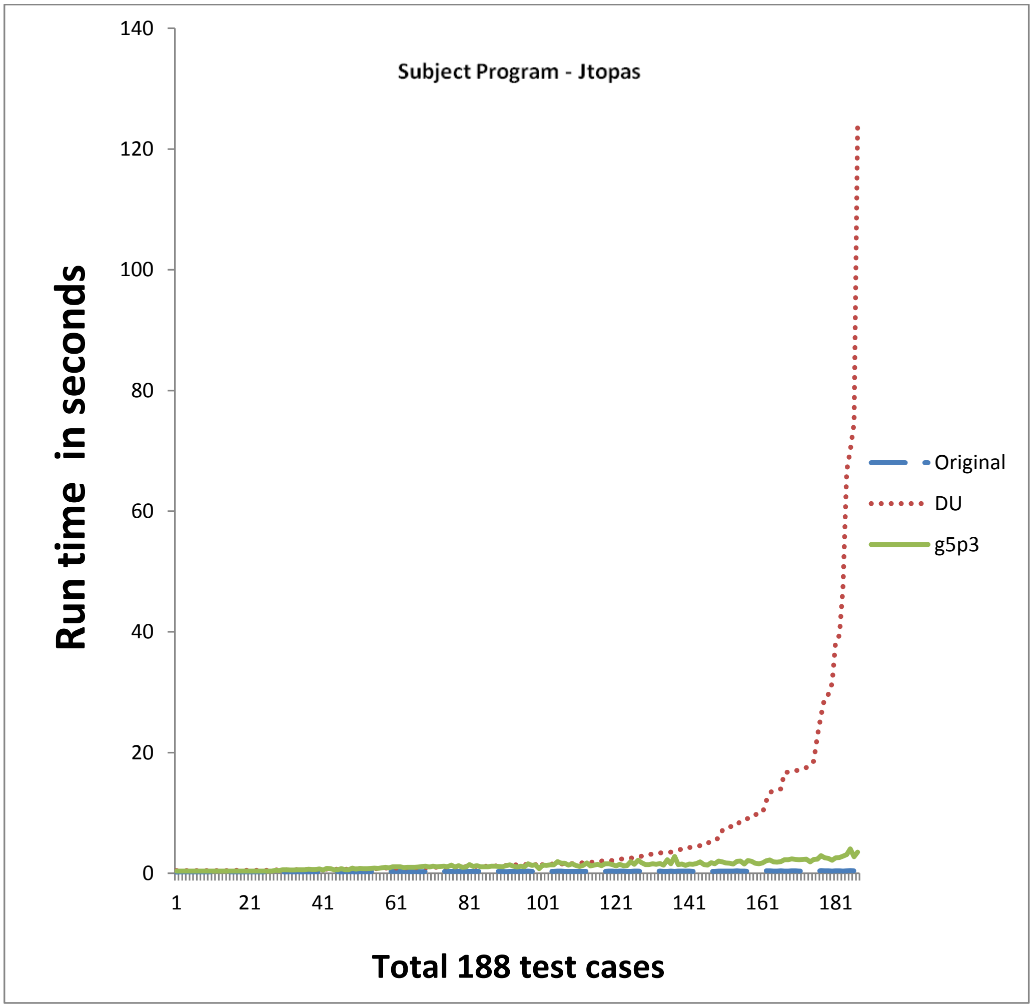


Figure 6.4: A sample run time for Def-Use (DU,) MPSC(g5,p3) and non-instrumented versions of all test cases in order, subject program `Jtopas`.

Table 6.2: Percentage of Missing information.

Subject Program	Missing information(%)
`jtopas`	0.0321
`nanoxml`	0.067
`xml-sec`	0.0368
`tcas`	0.0754
`scimark2`	0.0558

6.4 Accuracy of Bitset Implementation

As mentioned above, the bitset implementation may lose information about how many tuples have been covered, due to hash collisions. We therefore studied the question of how much information was lost.

For each subject program, each setting of g and p, and each test case, we collected the number of MPSC tuples reported as covered by the hash set and the bitset implementation. For each subject program, we then calculated the percentage decrease in number of tuples reported as covered, as an average across all test cases and all settings of g and p.

We found that the average loss of accuracy ranged from 0.0321% to 0.0754% for all programs (see Table 6.2), or less than 1 out of 1000 tuples lost due to hash collisions. This result indicates that the bitset implementation could be useful in situations where high accuracy is not needed.

In Figure 6.5 we show information loss due to bitset implementation for different MPSC for the subject program `Nanoxml`. We observed no significant information loss across different MPSC; no significant pattern seems to exist. Similar graphs that showed random information loss resulted from other subject programs, suggesting that the rate of hash collisions is unrelated to g or p.

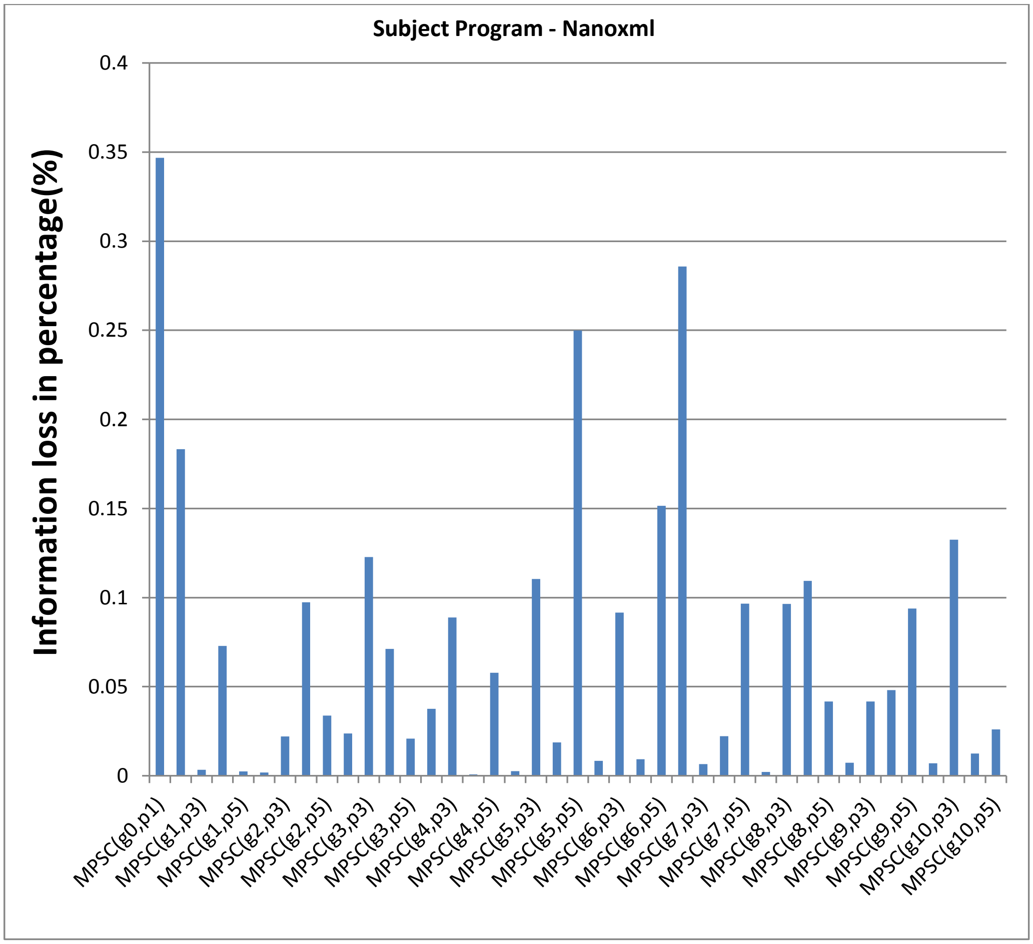


Figure 6.5: Information loss for different kinds of MPSC, subject program `Nanoxml`.

6.5 Summary

- MPSC usually incurs less overhead than DU.
- Different types of MPSC incur different amounts of overhead. It seems there is no correlation between incurred overhead with its corresponding *g* or *p* values.
- Overhead for longer test cases is significantly higher for DU than MPSC.
- We tested both hashtable and bitset implementation of MPSC, and the bitset implementation consistently showed greater efficiency accompanied by minor information loss.
- In bitset implementation, there is no pattern to the information loss: hash collisions are unrelated to *g* or *p* of MPSC.

Chapter 7

Discussion

Software engineering once lacked sufficient experimentation [92]. Software is developed rather than produced or manufactured [12], thus making experimentation in software engineering more challenging. Tichy et al.[91] showed that half of the published works on new design and models had no quantitative evaluation. In a different study Zelkowitz and Wallace also concluded the same [106, 107]. Most of the prominent code coverage criteria were proposed before the 1990's; there was great detail on how those criteria work but little empirical experimentation to show their effectiveness. In the last two decades, the amount of experimentation on code coverage increased, but few significant coverage criteria were introduced. In our study we not only introduce a new criterion, but also conduct several experiments to validate it.

Empirical validation methods can be quantitative or qualitative; experimentations fall into the quantitative category [60], where some specific procedure is followed to produce numerical data which are analyzed statistically to get deeper insight. We performed a multi-project [14] multi-lingual experiment. Basili [13] suggests that introducing a new model should follow a revolutionary research paradigm; a new model should be analyzed through experimentation to understand and measure its properties to reveal how and why it might be useful in some certain circumstances.

Software engineering researchers are often criticized for not explicitly describing their research paradigm [86]. Therefore, we will justify the design of our experiments in the following sections. We will also discuss some interesting observations about branch coverage which were a byproduct of our study.

7.1 Experiment structure

Our study had two main research goals: understanding MPSC and checking its relevance in software testing. We have designed two experiments for this purpose (see Figure 7.1). Furthermore, we have come up with four specific research questions whose answers can fulfill our research goals. In previous chapters we discussed those research questions and related experimental parts with result and analysis. Now in the following sections we will discuss the structure and semantics of these two experiments.

7.1.1 Understanding MPSC

We planned an experiment to understand the characteristics of MPSC. We designed an inductive experiment where we observed different properties of MPSC and created mathematical models to describe them using empirical analysis. The research question that guided us in this part of the study is:

- **RQ1.** How many MPSC tuples typically need to be collected for a program, and how is that number related to other program metrics?

Beyond creating a clear mathematical definition of MPSC, the major challenge was to find metrics that were precisely measurable and representative of MPSC. Individually a test case can behave erratically, so we not only defined metrics that represent a test case, but also some metrics that represent test suites of different sizes. In chapter 2 we describe them in detail.

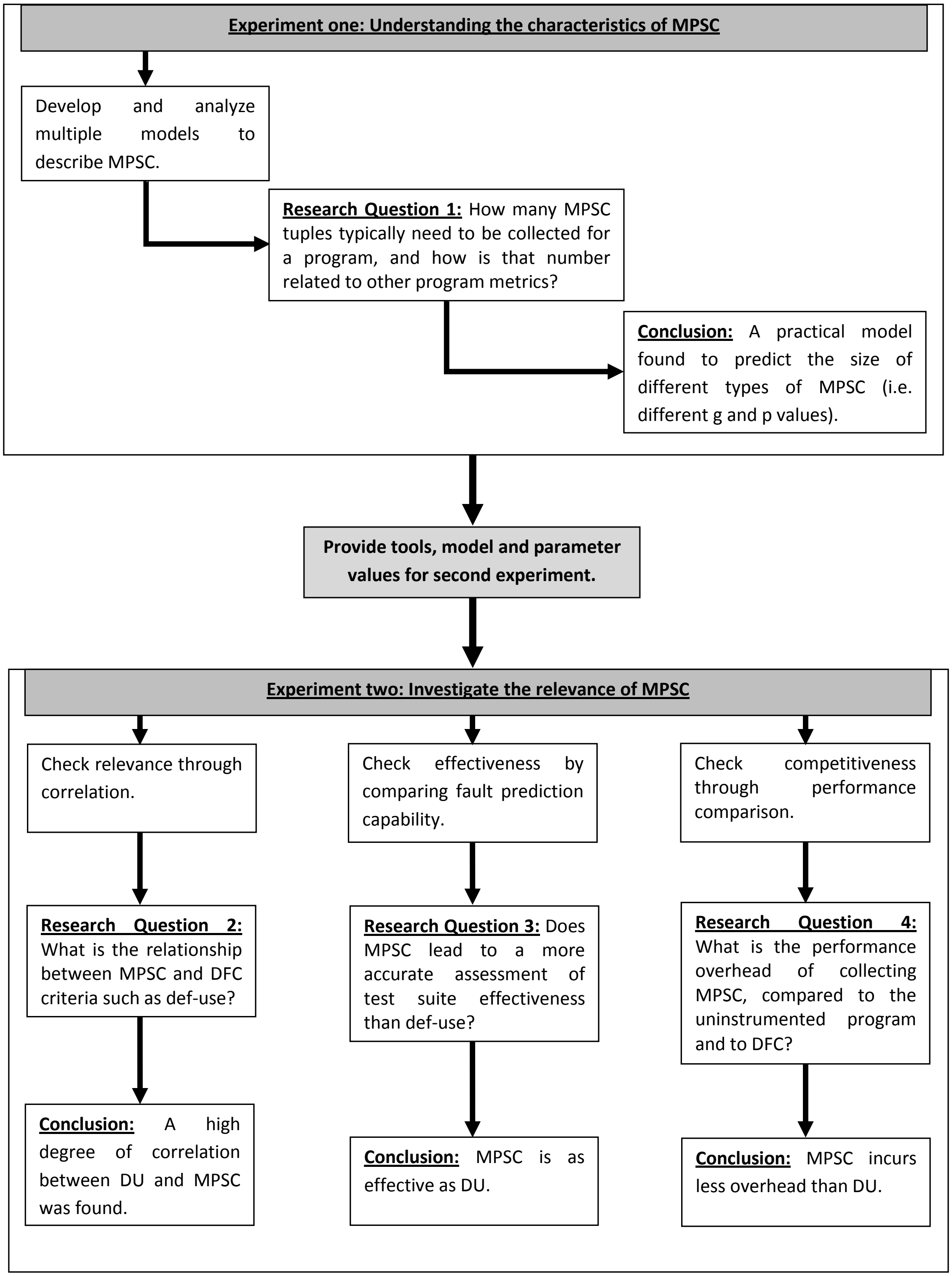

Experiment one: Understanding the characteristics of MPSC
Develop and analyze multiple models to describe MPSC.
Research Question 1: How many MPSC tuples typically need to be collected for a program, and how is that number related to other program metrics?
Conclusion: A practical model found to predict the size of different types of MPSC (i.e. different g and p values).
Provide tools, model and parameter values for second experiment.
Experiment two: Investigate the relevance of MPSC
Check relevance through correlation.
Check effectiveness by comparing fault prediction capability.
Check competitiveness through performance comparison.
Research Question 2: What is the relationship between MPSC and DFC criteria such as def-use?
Research Question 3: Does MPSC lead to a more accurate assessment of test suite effectiveness than def-use?
Research Question 4: What is the performance overhead of collecting MPSC, compared to the uninstrumented program and to DFC?
Conclusion: A high degree of correlation between DU and MPSC was found.
Conclusion: MPSC is as effective as DU.
Conclusion: MPSC incurs less overhead than DU.

Figure 7.1: Experiment structure

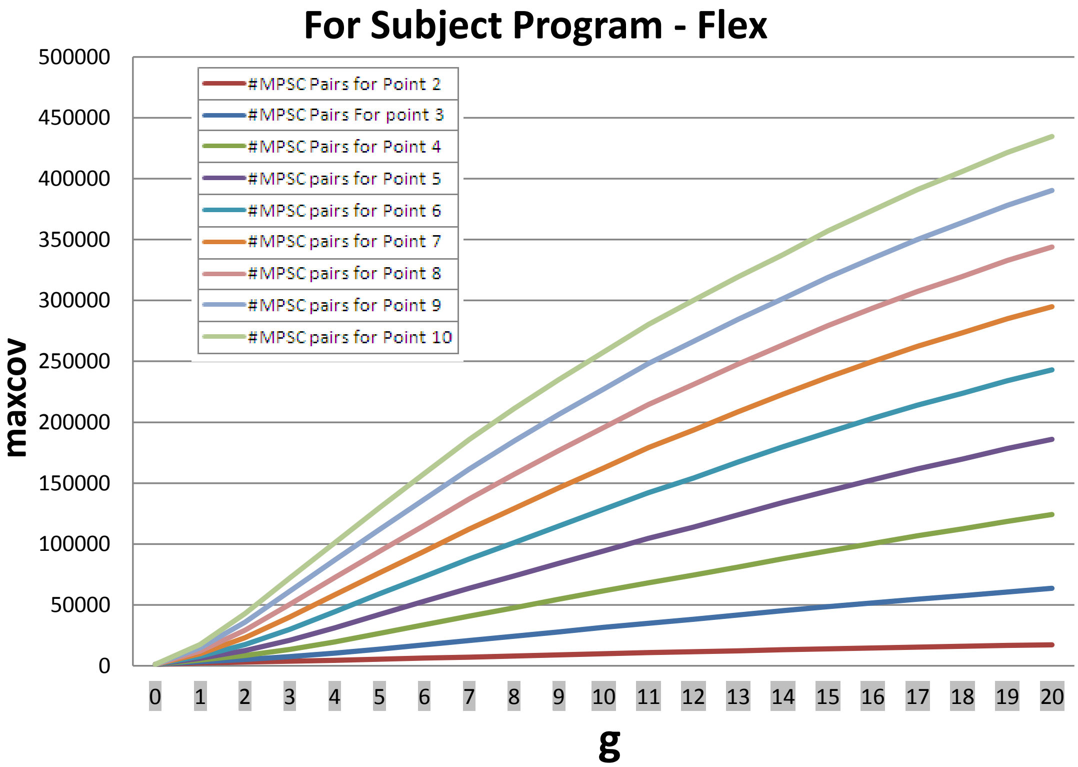


Figure 7.2: A sample `maxocv` graph for larger MPSC, subject program `Flex`

We have developed some tools to collect those metrics in a feasible way. Variation of MPSC can be very large and infeasible to collect due to resource constraints [1], so we only collected data for all MPSC where $g <= 10$ and $p <= 5$. Some of our early efforts with larger MPSC suggest that observable patterns are consistent for higher g and p (see Figure 7.2). We used statistical procedures to analyze the collected data and extract some models which can represent MPSC for practical use. Some of those models were necessary for our second experiment.

[1] A test case traversing N features (i.e. branches) sequentially can create MPSC for $g = 0\ to\ N - 1$ and $p = 2\ to\ N$. For some subject programs like *Jtopas*, N goes up to three million for some test cases.

7.1.2 Relevance of MPSC in software testing

After understanding the characteristics of MPSC and creating some models to describe them, we have to check its relevance in software testing. We checked it in three phases: through observing statistical correlation, checking effectiveness, and analyzing performance. We primarily use def-use for comparison as we indicate in Chapter 1 that our primary goal was to find a suitable coverage criterion as effective as DU but less complex to implement and incurring less overhead.

Check relevance through correlation

We mentioned in chapter 3 that MPSC is not subsumed by def-use, but both of them are collected from the same program using the same test suites. So some form of relationship may exist if we analyze them statistically. The related research question is as follows:

- **RQ2.** What is the relationship between MPSC and DFC criteria such as def-use?

We observed different correlations that derived from Pearson, Spearman and Kendall for DU and MPSC on random sets. It showed a high degree of correlation exists between DU and MPSC, especially for smaller g and p.

Fault prediction capability

Generally, when code coverage is used as a tool for bench-marking test suites, higher coverage of a suite should allow for more comprehensive fault-detection capability. We investigated the fault-detection capability of MPSC and compared it with DU using some empirical models. The related research question is as follows:

- **RQ3.** Does MPSC lead to a more accurate assessment of test suite effectiveness than def-use?

Our experiment using the mutant detection approach suggests MPSC is as good as DU. It is possible to compare many different types of MPSC with an ease which is intriguing.

Performance Analysis

Our earlier experiments were encouraging, so we wanted to analyze the performance of different types of MPSC for two distinct implementations. The related research question is as follows:

- **RQ4.** What is the performance overhead of collecting MPSC, compared to the uninstrumented program and to DFC?

Experience demonstrates that high overhead can make even a very strong coverage criterion impractical. DU did not attract sufficient enthusiasm from industry, possibly because of its high overhead cost and the complexity of its implementation. We found that MPSC incurs significantly less overhead than DU for both hash-table and bit-set implementation; for lengthier test cases, the difference is especially large.

7.2 Threats to Validity and their Mitigation

Sound empirical experimentation requires us to consider carefully threats to validity as these threats may influence experimental outcomes and limit our ability to interpret the results and thus conclude the study. Construct, internal and external validity must be protected from threats [76]. In the following sections, we will explain how we handle those threats.

7.2.1 Internal validity

We checked and visualized data and the results of statistical analyses in various ways. We collected data for $g = 0$ and several values of p to confirm that they were the same, in order to increase our confidence in our experimental procedures. The use of many randomly-chosen test suites was intended to increase internal validity.

7.2.2 Construct validity

Mutation adequacy of randomly constructed test suites has been shown to be a good measure of effectiveness, but other measures are possible. For subject programs with low-coverage test suites, we may have judged some mutants to be equivalent which were not; however, as noted below, our results were consistent across subjects.

7.2.3 External validity

Our results do not extend to other variants of MPSC, such as intraprocedural variants, or variants based on statement, block, or condition coverage. We hope to study such variants in the future. The subjects that we used may not be representative of real-world software. Our subjects have complementary strengths and weaknesses: the small Siemens programs have large, thorough test pools, and the programs with smaller, less thorough test pools are larger in SLOC. We are not aware of any subject programs that are larger than the Unix utilities we used here and also have test pools approaching the size and thoroughness of the Siemens programs. Our results are consistent across program sizes and test pool sizes, and across three related but distinct programming languages.

7.3 Def-use and Branch Coverage

Our most unexpected result was the high correlation of def-use coverage with branch coverage, and the lack of benefit of def-use coverage over branch coverage for predicting test suite effectiveness. The Hutchins et al. study [54] had led us to expect that def-use would distinguish between test suites more than branch coverage, and that def-use would be a better predictor of test suite effectiveness.

Our results, however, show that when test suite size is taken into account, def-use coverage performs very similarly to branch coverage. These results are consistent with those reported by Siami Namin and Andrews [87]. Together they may indicate that the improved effectiveness of high def-use coverage test suites in the Hutchins et al. study is an artefact of the fact that high def-use coverage test suites need to be bigger than high branch coverage test suites.

We should note that these results do not extend to other problems in software testing. Santelices et al., for instance, showed that DU coverage, and combinations of DU coverage with other forms of coverage, yielded more accurate fault localization than branch coverage [85]. More experimentation would be needed to show whether MPSC coverage can replace DU coverage for fault localization.

7.4 MPSC and Branch Coverage

We found that branch coverage was often the most accurate predictor of test suite effectiveness when combined with test suite size. However, for some subject programs, MPSC with some value of $g > 0$ and $p > 1$ was a better predictor. We also found that our instrumentation for MPSC was efficient even with $g > 0$ and $p > 1$.

These results taken together suggest that users seeking a coverage criterion stronger than decision coverage can use MPSC with $g = 1$ and high p. Since this form of coverage subsumes decision coverage, is usually more efficient to collect than def-use coverage, and sometimes yields a better prediction of test suite effectiveness than def-use coverage, it may present a better option than def-use coverage.

Chapter 8

Conclusions and Future Work

Previous research suggests that data flow coverage is better than common control flow coverage in terms of predicting faults, but less efficient as it incurs a higher overhead cost. Data flow coverage inherently captures partial execution sequences, possibly making it more effective than control flow coverage. Along with structural elements (that is, statements, branches, conditions and others) a body of code has a sequence which represents logical steps to perform a task. Popular control flow coverage measures such as statement or branch coverage do not consider execution sequence, possibly making them less effective than data flow coverage. However, a more versatile control flow coverage criterion such as path coverage which captures execution sequence, is infeasible even for a small program due to its path explosion problem.

We propose control flow coverage criteria that explicitly target the execution sequence as a part of the coverage definition but solve the explosion problem by limiting the length of the sequence. We called it Multi Point Stride Coverage (MPSC); we can get different variations of MPSC just by manipulating some simple parameters such as the number of execution points and the size of the gaps between them. Our approach suggests an easier and more efficient implementation with the lower overhead cost of control-flow-coverage criteria and the effectiveness of data-flow-coverage criteria.

In this thesis, we defined and presented the results of empirical studies on MPSC. We compared it to def-use coverage, a well-known data flow coverage criterion. In the following sections we will conclude our study and identify some potential future research areas where we may introduce MPSC.

8.1 Conclusions

We conducted experiments to understand the characteristics of MPSC and to investigate its potential for software testing. Our study suggests MPSC is a relevant and competitive coverage criterion with a high potential. Some important results from our study include the following points:

- **MPSC is predictable:** Control flow coverage such as branch or statement coverage is precisely countable [1] , making it possible to understand and compare the thoroughness of the test suites. We found that the maximum number of MPSC tuples that need to be collected is highly predictable, given a constant which is characteristic of a program. The predictably of MPSC allows for an efficient implementation and resource management while testing.

- **Coverages are statistically correlated:** We found that def-use coverage is strongly correlated with branch coverage and often does not yield a better prediction of test suite effectiveness than branch coverage when test suite size is taken into account. It is possible that dissimilar coverage criteria using the same program to collect defined features may cause this statistical correlation.

- **MPSC may be a better predictor of test effectiveness:** We also found that MPSC with $g > 0$ and $p > 1$ sometimes yielded a better prediction of effectiveness than branch coverage. The potential MPSC variations are huge which gives a wider range of choice; based on program characteristics one variation may predict more effectively than others.

[1] The count shows how many loops (i.e. for, while), condition structures (i.e. if, switch/case) or statements there are in a program. The same is true to some degree for data-flow coverage such as def-use, though it is more difficult to precisely collect, especially for dynamic memory.

- **MPSC is practical and efficient:** We developed prototype implementations of MPSC data-collection libraries that performed well compared to a state-of-the-art def-use coverage tool. Our experience suggests that even traditional coverage tools which preserve sequence while collecting features can be used to collect MPSC with relatively minor effort.

8.2 Future work

Future work includes improving the efficiency of our prototype implementations of MPSC and exploring variants of MPSC. We are also planning to apply MPSC in other problem domains in software testing. In the following we will give some brief descriptions of those domains and how MPSC can play some role:

- **Fault localization:** Fault localization should reveal the location of bugs in a program's code. Code coverage can play a vital role in locating faults. For example, Wong et al. [101] suggested some heuristics to use coverage information. They suggest that a portion of code thoroughly covered in a successful run has less chance of having bugs than rarely visited parts. They use statement coverage where we can use MPSC; the chance of having unique MPSC pairs (that is, fragments of less-visited paths) in the event of faults is higher than with simple statements.

- **Test suite prioritization:** In regression testing, insufficient resources make it impossible to re-execute all test cases in every new build, a major problem given the common practice of nightly builds among large software companies. Test case prioritization techniques help in this regard by ordering the test cases to maximize the benefits [61]. Code coverage plays a guiding role in this process and we can use MPSC for this purpose.

- **Test suite augmentation:** Through source code analysis it is possible to find all possible MPSC pairs for a particular variant of MPSC. We can use that information to augment test suites. Generated suites guided by MPSC have a better chance of capturing logical variations of the code than other simple coverage measures.

Bibliography

[1] Software considerations in airborne systems and equipment certification. Advisory Circular DO-178B, Radio Technical Commission for Aeronautics, 1992. Errata Issued 3-26-99.

[2] Guest editors' introduction: Software testing practices in industry. *Software, IEEE*, 23(4):19 –21, July-August 2006.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98, Windsor, United Kingdom, September 2007.

[4] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the accuracy of fault localization techniques. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 76–87, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Intl. Conf. Software Eng (ICSE)*, St. Louis, MO, May 2005. 402-411.

[7] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32:608–624, 2006.

[8] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Intl. Conf. Automated Software Eng. (ASE)*, pages 144–153, Atlanta, GA, 2007.

[9] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. *SIGPLAN Not.*, 40(6):201–212, 2005.

[10] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, pages 1–22, Leiden, The Netherlands, November 2004.

[11] Thomas Ball and James R. Larus. Efficient path profiling. *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-29)*, Dec 1996. Madison, WI.

[12] Victor R. Basili. The experimental paradigm in software engineering. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 3–12, London, UK, UK, 1993. Springer-Verlag.

[13] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 442–449, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *Software Engineering, IEEE Transactions on*, SE-12(7):733 –743, July 1986.

[15] Pete Behrens. Enterprise Agility II: Microsoft and IBM, November 2005. (Accessed on November 5, 2012).

[16] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[17] Stefan Berner, Roland Weber, and Rudolf K. Keller. Enhancing software testing by judicious use of code coverage information. In *Intl. Conf. Software Eng. (ICSE)*, pages 612–620, Minneapolis, MN, May 2007.

[18] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[19] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[20] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 220–233, New York, NY, USA, 1980. ACM.

[21] Jon Cargille and Barton P. Miller. Binary wrapping: a technique for instrumenting object code. *SIGPLAN Not.*, 27(6):17–18, June 1992.

[22] T. Chen, H. Leung, and I. Mak. Adaptive random testing. 3321:3156–3157, 2005.

[23] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[24] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, SE-4(3):178–187, 1978.

[25] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Intl. Conf. Software Eng (ICSE)*, pages 71–80, Leipzig, Germany, May 2008.

[26] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[27] J. Delange, J. Hugues, and L. Pautet. Couverture: an innovative open framework for coverage analysis of safety critical applications. *Ada User Journal*, 30(4):248 –272, December 2009.

[28] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34 –41, April 1978.

[29] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[30] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[31] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[32] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Software Eng.*, 32(9):733–752, 2006.

[33] Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, *Computational Science – (3rd ICCS'03, Part IV)*, volume

2660 of *Lecture Notes in Computer Science (LNCS)*, pages 53–62. Springer-Verlag (New York), Saint Petersburg, Russia / Melbourne, Australia, June 2003.

[34] K J Ellis and R G Duggleby. What happens when data are fitted to the wrong equation? *Biochem J.*, 171(3):513517, 1978.

[35] Emelie Engström and Per Runeson. Software product line testing - a systematic mapping study. *Information and Software Technology*, 53(1):2 – 13, 2011.

[36] P.G. Frankl and S.N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Software Engineering, IEEE Transactions on*, 19(8):774 –787, August 1993.

[37] Phyllis Frankl and Elaine Weyuker. Provable improvements on branch testing. *IEEE Trans. Software Eng.*, 19(10), October 1993.

[38] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 154–164, New York, NY, USA, 1991. ACM.

[39] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 313–319, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[40] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Not.*, 10(6):493–510, 1975.

[41] James R. Goodman. Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News*, 11(3):124–131, June 1983.

[42] J. P. Guilford. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, New York, 1956.

[43] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4 –12, 2002.

[44] R. Hamlet. Theoretical comparison of testing methods. *SIGSOFT Softw. Eng. Notes*, 14(8):28–37, November 1989.

[45] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. Nasa/tm-2001-210876, 2001.

[46] MatsP.E. Heimdahl and Devaraj George. On the effect of test-suite reduction on automatically generated model-based tests. *Automated Software Engineering*, 14:37–57, 2007.

[47] M.P.E. Heimdahl and D. George. Test-suite reduction for model based tests: effects on test quality and implications for testing. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 176 – 185, sept. 2004.

[48] M.P.E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 178 – 186, march 2004.

[49] J. R. Horgan and S. A. London. Data flow coverage and the C language. In *Symp. on Testing, Analysis, and Verification (TAV)*, pages 87–97, 1991.

[50] J.R. Horgan and S.A. London. ATAC: A data flow coverage testing tool for C. In *Symp. on Assessment of Quality Software Development Tools*, pages 2 – 10, May 1992.

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

[52] W.E. Howden. Functional program testing. *Software Engineering, IEEE Transactions on*, SE-6(2):162 – 169, March 1980.

[53] Pei Hsia, David Kung, and Chris Sell. Software requirements and acceptance testing. *Annals of Software Engineering*, 3:291–317, 1997. 10.1023/A:1018938021528.

[54] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Intl. Conf. Software Eng. (ICSE)*, pages 191–200, Sorrento, Italy, May 1994.

[55] R. Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling*. John Wiley and Sons, Inc., New York USA, 1991.

[56] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. A novel method of mutation clustering based on domain analysis. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE09)*, pages 422–425. Knowledge Systems Institute Graduate School, 2009.

[57] Yue Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 249 –258, Sept. 2008.

[58] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649 –678, September-October 2011.

[59] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Intl. Conf. Software Eng. (ICSE)*, pages 467–477, Orlando, FL, May 2002.

[60] Heiko Koziolek. The role of experimentation in sofware engineering. In *Trustworthy Software Systems*, volume 1. Wilhelm Hasselbring, 2005.

[61] Z. Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on*, 33(4):225 –237, April 2007.

[62] Brian Marick. The weak mutation hypothesis. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 190–199, New York, NY, USA, 1991. ACM.

[63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[64] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Glenford J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[66] Nachiappan Nagappan, E. Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13:289–302, 2008. 10.1007/s10664-008-9062-z.

[67] Simeon C. Ntafos. On required element testing. *IEEE Trans. Software Eng.*, SE-10(6):795–803, November 1984.

[68] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[69] A. Jefferson Offutt, Roy P. Pargas, Scott V. Fichter, and Prashant K. Khambekar. Mutation testing of software using a MIMD computer. In *in 1992 International Conference on Parallel Processing*, pages 257–266, 1992.

[70] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 100–107, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[71] A. Jefferson Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[72] Thomas Ostrand. *White-Box Testing*. John Wiley & Sons, Inc., 2002.

[73] Carlos Pacheco, Shuvendru K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Intl. Conf. Software Eng. (ICSE)*, pages 75–84, Minneapolis, MN, May 2007.

[74] Ron Patton. *Software Testing*. Sams, Indianapolis, IN, USA., 2005.

[75] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *21st International Conference on Software Engineering (ICSE '99)*, pages 277–284, May 1999.

[76] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA, 2000. ACM.

[77] Simone Pimont and Jean-Claude Rault. A software reliability assessment based on a structural and behavioral analysis of programs. In *Intl. Conf. Software Eng. (ICSE)*, pages 486–491, San Francisco, CA, 1976.

[78] Ajitha Rajan, Michael W. Whalen, and Mats P.E. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Intl. Conf. Software Eng. (ICSE)*, pages 161–170, Leipzig, Germany, 2008.

[79] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, SE-11(4):367–375, April 1985.

[80] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[81] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Intl. Conf. Software Maintenance (ICSM)*, pages 34–43, Washington, DC, November 1998.

[82] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22 –29, July-Aug. 2006.

[83] Raúl Santelices and Mary Jean Harrold. Efficiently monitoring data-flow test coverage. In *Intl. Conf. Automated Software Eng. (ASE)*, pages 343–352, November 2007.

[84] Raul Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 195–206, New York, NY, USA, 2010. ACM.

[85] Raúl Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *Intl. Conf. Software Eng. (ICSE)*, pages 56–66, 2009.

[86] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer (STTT)*, 4:1–7, 2002. 10.1007/s10009-002-0083-4.

[87] Akbar Siami Namin and James Andrews. The influence of size and coverage on test suite effectiveness. In *Intl. Symp. Software Testing and Analysis (ISSTA)*, pages 57–68, Chicago, IL, 2009.

[88] Ben Smith and Laurie Williams. A survey on code coverage as a stopping criterion for unit testing. Technical Report TR-2008-22, Department of Computer Science, North Carolina State University, Sep 2008.

[89] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, July 2002.

[90] D. Talby, A. Keren, O. Hazzan, and Y. Dubinsky. Agile software testing in a large-scale project. *IEEE Software*, 23(4):30 –37, July-Aug. 2006.

[91] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9 – 18, 1995.

[92] W.F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32 –40, May 1998.

[93] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.

[94] N. Tillmann and W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38 –47, July-Aug. 2006.

[95] Jay Trimble and Chris Webster. Agile development methods for space operations. In *The 12th International Conference on Space Operations*, 2012.

[96] Ramtilak Vemu and Jacob A. Abraham. Budget-dependent control-flow error detection. In *International On-Line Testing Symposium*, pages 73–78. IEEE, 2008.

[97] W. N. Venables, D. M. Smith, and The R Development Core Team. An introduction to R. Technical report, R Development Core Team, June 2006.

[98] E.J. Weyuker. Can we measure software testing effectiveness? In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 100 –107, May 1993.

[99] E.J. Weyuker. Testing component-based software: a cautionary tale. *Software, IEEE*, 15(5):54 –59, September/October 1998.

[100] L. Williams, G. Kudrjavets, and N. Nagappan. On the effectiveness of unit test automation at Microsoft. In *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, pages 81 –89, November 2009.

[101] W.E. Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 449 –456, July 2007.

[102] W.Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.

[103] Martin R. Woodward, David Hedley, and Michael A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Software Eng.*, SE-6(3):278–286, May 1980.

[104] Martin R. Woodward and Michael A. Hennell. On the relationship between two control-flow coverage criteria: All JJ-paths and MCDC. *Information and Software Technology*, 48:433–440, 2006.

[105] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5), 2009.

[106] Marvin V. Zelkowitz and Dolores Wallace. Experimental validation in software engineering. *Information and Software Technology*, 39(11):735 – 743, 1997.

[107] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23 –31, May 1998.

[108] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *Software Engineering, IEEE Transactions on*, 22(4):248 –255, April 1996.

[109] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29:366–427, December 1997.

Appendix A

JQXZ: A tool to instrument MPSC coverage

We have developed a tool called JQXZ for instrumenting source code to get MPSC coverage information. JQXZ has four main components: Tokenizer, Instrumentation inserter, Un-tokenizer and JQXZ library. We give a short description of them below:

- **Tokenizer:** This tool tokenizes any C, C++ or Java code file. It was written in C. The purpose is to tokenize the original source code file for easier manipulation.

- **Instrumentation Inserter:** It accepts a tokenized file and modifies it to add instrument for all branch tokens (i.e. if, while, for and switch case). Basically it inserts a function call with two parameters, a condition expression and a unique identifier for each branch. It also inserts some header information (for example an import statement for Java) necessary to use JQXZ's library. All stabs are inserted in token form. This tool was written in Java.

- **Un-tokenizer:** It converts the tokenized file to a plain code file. It was also written in C.

- **JQXZ library:** We support three languages: C, C++ and Java. So we wrote three different modules each for C, C++ and Java respectively to record MPSC coverage information. They contain definitions for those functions which were inserted in the source code instrumentation phase. Those modules are called at execution time to report any branch traversal.

- **JQXZ Manager:** We wrote some shell scripts to manage different parts of JQXZ. The scripts search code files, categorize them based on programming language and then subsequently use the above three tools accordingly to instrument them for MPSC. The manager also supports the un-instrumentation process.
- **MPSC Analyzer:** We also wrote many small tools in UNIX shell, R, C and Java for extracting data from traces and analyzing them.

In Figure A.1 we show the instrumentation process. The process transforms a simple source-code file into an instrumented code file. Due to this instrumentation, after compilation and build, the software gains the capability to report MPSC coverage information without changing its operational behavior.

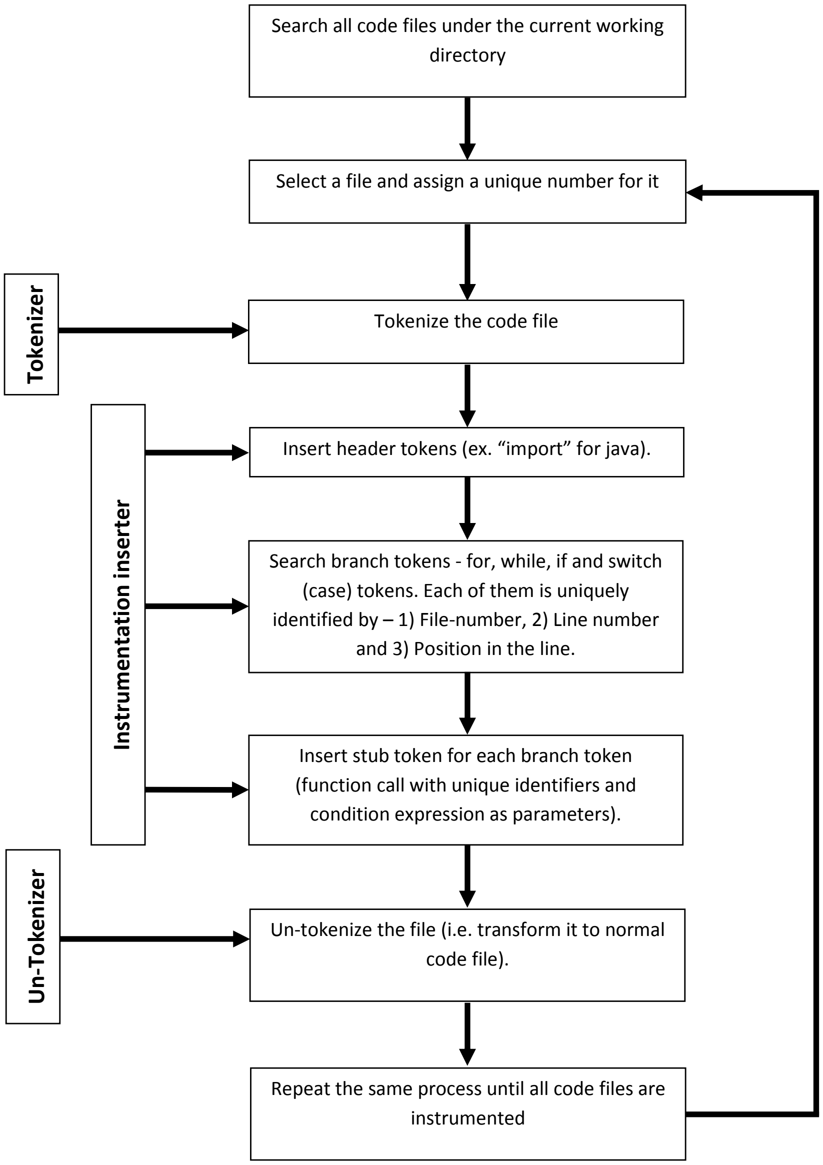

Search all code files under the current working directory
Select a file and assign a unique number for it
Tokenizer
Tokenize the code file
Instrumentation inserter
Insert header tokens (ex. “import” for java).
Search branch tokens - for, while, if and switch (case) tokens. Each of them is uniquely identified by – 1) File-number, 2) Line number and 3) Position in the line.
Insert stub token for each branch token (function call with unique identifiers and condition expression as parameters).
Un-Tokenizer
Un-tokenize the file (i.e. transform it to normal code file).
Repeat the same process until all code files are instrumented

Figure A.1: Source code instrumentation process for MPSC coverage by JQXZ

Appendix B

Subject Program Biographies

Except concordance all the subject programs we used in our experiments were downloaded from SIR (Software Infrastructure repository) [1]. In their site they provide some biographical description of those subject programs except “sed”. There is no formal description of sed, according to summary information in SIR, it’s a “Linux patches contributed by Ben Liblit, University of Wisconsin”.

The following is taken from SIR, except for “concordance” which is adapted from the published paper of Siami Namin and Andrews [87]. We only remove those program names which we did not use in our experiments.

B.1 Biographies of flex, grep, gzip

Flex, grep, gzip are all unix utilities obtained from the Gnu site. We obtained several sequential, previously-released versions of each of these programs.

For these objects, with the exception of a few “smoke” tests, comprehensive test suites were not available. To construct test cases representative of those that might be created in practice for these programs, we used the documentation on the programs, and the parameters and special effects we determined to be associated with each program, as informal specifica-

[1] http://sir.unl.edu/portal/

tions. We used these informal specifications, together with the category partition method and an implementation of the TSL tool, to construct a suite of test cases that exercise each parameter, special effect, and erroneous condition affecting program behavior. We then augmented those test suites with additional test cases to increase code coverage (measured at the statement level). We created these suites for the base versions of the programs; they served as regression suites for subsequent versions.

We wished to evaluate the performance of testing techniques with respect to detection of regression faults, that is, faults created in a program version as a result of the modifications that produced that version. Such faults were not available with our object programs; thus, to obtain them, we followed a procedure similar to one defined and employed in several previous studies of testing techniques, as follows. First, we recruited graduate and undergraduate students in computer science with at least two years of C programming experience. Then, the students thus recruited were instructed to insert faults that were as realistic as possible based on their experience, and that involved code deleted from, inserted into, or modified between the versions. To further direct their efforts, the fault seeders were given a list of types of faults to consider.

Given ten potential faults seeded in each version of each program, we activated these faults individually, and executed the test suites for the programs to determine which faults could be revealed by which test cases. We excluded any potential faults that were not detected by any test cases: such faults are meaningless to our measures and have no bearing on results. We also excluded any faults that were detected by more than 25% of the test cases; our assumption was that such easily detected faults would be detected by engineers during their unit testing of modifications.

B.2 Biographies of Siemens subject program

The "Siemens" programs were assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [54], and were made available to us by Tom Ostrand. They have since been partially modified by us for use in further studies.

The Siemens programs perform a variety of tasks: tcas is an aircraft collision avoidance system, schedule2 and schedule are priority schedulers, totinfo computes statistics given input data, printtokens and printtokens2 are lexical analyzers, and replace performs pattern matching and substitution.

The researchers at Siemens sought to study the fault detecting effectiveness of coverage criteria. Therefore, they created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the program. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working mostly without knowledge of each other's work. The result of this effort was between 7 and 41 versions of each base program, each containing a single fault.

For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. To populate these test pools, they first created an initial suite of black-box test cases according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code, using the category partition method and the Siemens Test Specification Language tool. They then augmented this suite with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test cases. To obtain meaningful results with the seeded versions of the programs, the researchers retained only faults that were neither too easy nor too hard to detect, which they defined as being detectable by at most 350 and at least 3 test cases in the test pool associated with each program.

To obtain sample test suites for these programs, we used the test pools for the base programs and sampling procedures to create suites. We have created several types of suites, some randomly selected, some coverage adequate. The Siemens CONTENTS files describe the types of suites.

The Siemens files are described in the original Siemens paper [54].

B.3 Biography of jtopas

JTopas is a Java library used for parsing text data, available at http://jtopas.sourceforge.net/jtopas. The JTopas classes and interfaces in their current state of development can be used for tokenizing and basic parsing tasks. A command line parser, a file reader, a IP protocol interpreter, a partial HTML parser or a tokenizer for JavaCC/JTB may be realized with JTopas. This flexibility is achieved by dynamically configurable classes and strict separation of different tasks.

B.4 Biography of XML-security

XML-security is a component library implementing XML signature and encryption standards, supplied by the XML subproject of the open source Apache project. It is available at http://xml.apache.org/secu
It currently provides a mature implementation of Digital Signatures for XML, with implementation of encryption standards in progress.

We obtained several sequential, previously-released versions of XML-security, each provided with a developer supplied JUnit test suite. In each version, faults were seeded using the fault seeding procedure described in the Java Fault Seeding Process.

B.5 Biography of nanoxml

NanoXML is a small XML parser for Java, available at http://nanoxml.sourceforge.net/orig. NanoXML is an easy-to-use, non-GUI based, freely available system, buildable from source without using external libraries.

The extensible markup language, XML, is a way to mark up text in a structured document. It is designed to improve the functionality of the web by providing more flexible and adaptable information identification.

NanoXML is a component library, so we have also found an application, JXML2SQL, available with NanoXML, that uses NanoXML. JXML2SQL takes as input an XML file and either transforms it into an html file, showing the contents in tabular form, or into an SQL file.

Once we had the six versions of NanoXML, and the application that uses it, we created tests for it using the category partition method. We created these for the externally visible components of the NanoXML library, and also for the application. We created a first version of this suite for the base versions, then modified the suite as needed to handle changes in subsequent versions; this included adding some new tests.

B.6 Biography of concordance

The subject program `concordance` is a utility for making concordances (word indices) of documents. The original program was written in C++ by Ralph L. Meyer, and made available under the GNU General Public Licence on various open-source websites.The source file consists of 966 net lines of C++ code, comprising five classes and 39 function definitions.

Over the course of several years, Andrews and several classes of fourth-year students of a testing course at the University of Western Ontario identified 13 separate faults in the program. The source code was concatenated into one source file, and the faults were eliminated in such a way that they could be independently re-introduced to produce 13 faulty versions.

Six students in the testing course class of 2005 donated their test suites for `concordance`. These test suites were constructed according to guidelines for black-box testing (category-partition, boundary-value, extreme-value, syntax testing) and white-box testing (all test suites achieve maximum feasible decision/condition coverage). These test suites were put together to form a test pool with a total of 372 test cases.

Curriculum Vitae

Name: Mohammad Mahdi Hassan

Contact: mmmhbd76@gmail.com

Education

Ph.D. (Software Engineering)
University of Western Ontario, February 2013
London, ON, Canada

M.Sc. (Computer Science)
King Fahd University of Petroleum and Minerals, April 2007
Dhahran, Saudi Arabia

B.Sc. (Computer Science and Engineering)
Khulna University, December 1999
Khulna, Bangladesh

Honours and Awards:

Ontario Graduate Scholarship (OGS)
2011-2012

Related Work Experience:

Teaching Assistant
The University of Western Ontario, 2009 - 2012

Teaching Assistant
King Fahd University of Petroleum and Minerals, 2005 - 2007

Lecturer
King Fahd University of Petroleum and Minerals, 2007 - 2008

Lecturer
Asian University of Bangladesh, 2001