

Testing with Respect to Concerns

Amie L. Souter
Department of Computer Science
Drexel University
Philadelphia, PA 19104
souter@cs.drexel.edu

David Shepherd and Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
shepherd, pollock@cis.udel.edu

Abstract

Often the code regions that are assigned for a maintenance task do not follow the modularization of the original application program, but instead include parts of code from many different units scattered throughout the application. In this paper, we investigate an approach to testing which we call concern-based testing, which leverages existing tools to help software maintainers identify the relevant code for their assigned task, their concern. The main contribution is a demonstration of the possible savings in test suite execution overhead and the increased precision in coverage information that can be obtained for a software maintainer if testing tasks are performed with respect to concerns. Based on a concern graph representation of the concern, a framework for guiding selective instrumentation for scalable coverage analysis is also presented.

1. Introduction

Maintenance tasks often are assigned according to features or concerns that are not contained completely in local regions of the program, but instead span the developer's modularization of the program. Tools such as `grep` [1], code browsers [23, 15, 16, 25, 18, 27, 21], cross-reference databases [5], program slicers [32], and feature exploration tools based on a graphical representation of the program [27] all exist to help programmers identify the scattered code that is involved in a particular feature. This paper investigates the application of such tools for software testing. In particular, we propose using such tools to enable concern-based testing for scalability and improved coverage information during maintenance.

In this paper, a *concern* refers to a part of the application that is relevant to a particular maintenance task. One concern might be error handling. A particular concern is crosscutting when the code that implements the single concern includes parts of code from different methods, classes,

or components. Testing with respect to a concern that spans programmer-defined modularization is different from unit testing which focuses on testing programmer-defined units in isolation, such as procedures in imperative programs or classes in object-oriented programs. One might claim that integration testing is crosscutting in terms of testing the integration of units of modularization; however, concern-based testing is defined to include not only the interactions between units, but also subregions of the units that are part of the implementation of a particular concern.

Maintenance tasks can be categorized as corrective, adaptive, perfective, or preventive. While the motivation behind performing maintenance tasks varies, they all require a software maintainer to determine the portion of source code related to the task. While determining the code associated with the task, the maintainer could also select a set of test cases associated with the task. These test cases could be utilized to better understand the interaction between regions of a program, to perform coverage analysis, and to determine portions of the concern that have not been thoroughly tested during development.

By identifying the code associated with a concern for a particular maintenance task, only the associated code needs to be instrumented to determine coverage of the concern of interest. The space requirements for instrumentation can thus be reduced to be proportional to the size of the code implementing the concern instead of the whole program. When the instrumented program is run for coverage information, the running time should be reduced, with respect to executing a fully instrumented program. In addition, with the use of an abstract representation of the structure of the concern, hybrid forms of instrumentation can be enabled, where different parts of the concern are instrumented in different ways.

If test cases can be identified with respect to their coverage of particular concerns for maintenance, then the test suite can be organized according to concerns. This organization can provide quick selection of test cases for different concerns, and enable prioritization of test cases based on

the concern of interest. This prioritization can be used to decrease test execution time. Thus, one problem to be addressed is: Given a large program P , a test suite T , and a maintenance task focusing on concern C that spans the developer's modularization units for P , identify the subset S of T that covers code implementing C . The subset S will be the higher priority test cases for concern C during maintenance, as S excludes any test cases that do not cover any of C .

By answering this question, we also gain information about the percent coverage of individual concerns in the code. Some concerns may have higher coverage than others; this is not reflected in the coverage information reported at the system or unit scope. Furthermore, more precise information about the parts of a concern that are not currently covered by the test suite can be identified, directing the construction of additional test cases for a particular concern. In this way, testing can be made more scalable, by focusing on subsets of the whole test suite according to concerns, which represent the crosscutting regions of the code of interest to different software maintenance tasks, rather than the developer's modularization of the original program, which typically does not reflect the perspective of the code for maintenance.

The next section presents a case study of three concerns for a single application. In the case study, we explore some of the questions raised with respect to concern-based testing. Section 3 describes a graphical representation of a concern which enables selective and hybrid instrumentation as described through a parameterized framework. In Section 4, we discuss related work in priority-based and regression testing, concern location, and aspect-oriented programming. Section 5 provides a summary of our conclusions and plans for future research.

2. Motivating Case Study

This case study is intended to provide evidence of the advantages of utilizing a concern-based testing approach for maintenance activities. In particular, we explore several questions about potential benefits of focusing testing to a concern of interest during maintenance. The following questions were investigated:

Question 1: What space savings can be obtained by instrumenting a concern of interest versus a whole program?

Question 2: What test suite reduction can be obtained by identifying only those test cases associated with a concern?

Question 3: What is the difference in coverage obtained over a concern of interest versus other units of testing?

Our approach is based on the following steps. First, a user identifies a concern of interest for maintenance. There exist several different approaches to help software maintainers locate scattered code, including program slicers [11] and code browsers [23, 15, 16, 25, 18, 27, 21]. For this study, we utilized FEAT [14], an Eclipse plug-in [12], for locating concerns. Once a concern has been constructed, we utilize the concern to guide instrumentation of the source code. After the program has been instrumented, we can run a test suite to obtain concern-based coverage information. Such coverage information provides the ability to select a subset of test cases for further testing with respect to the concern and to gather concern-based coverage information.

We utilized the Jakarta `regex` package for this case study, which is a utility that provides an interface for describing and matching regular expressions [26]. The package consists of 2850 non-commented lines of source code, and is composed of 15 classes, 111 methods, and is accompanied by 174 test cases. We selected three concerns for this study. The following provides descriptions and characteristics of the three concerns. Figure 1 summarizes the characteristics.

Concern 1: Clustering

The first concern we located is based on the notion of clusters. A regular expression in which portions of a pattern are grouped together into subpatterns is called a cluster. For example, when specifying a regular expression, parenthesis can be used to group pieces of the overall regular expression. For instance, the expression $(a|b)(c|d)$ is separated into two parts, $(a|b)$ and $(c|d)$. Commonly, capturing and clustering are used within regular expressions. Capturing will save information regarding each subexpression where clustering will not.

Clustering is a concern integrated into the `RECompiler` and `RE` classes of the `regex` package. While the clustering concern is relatively small, accounting for only 36 lines of source code and 2 methods in the `regex` package, it is actually a newly added feature to the package. Previously, only capturing was supported.

Concern 2: Error Handling

The *Error Handling* concern represents source code in the program that deals with error conditions within the program. Such error handling includes checking for syntax errors within a regular expressions, string bounds checking, and internal error conditions. This concern is a good demonstration of the concept of a crosscutting concern, since it is scattered across 5 classes, 11 methods, and 201 lines of source code.

Characteristics	Whole Program	Concerns		
		Clustering	Error Handling	Matching
Lines of code	2850	36	201	760
Number of classes	15	2	5	1
Number of methods	111	2	11	5
Space for instrumented code	100K	51K	56K	52K
% reduction in test suite	–	58%	82%	0%

Table 1. Concern characteristics.

Concern 3: Matching a regular expression

Matching a regular expression to a given input string is the basis for the matching concern. The *Matching* concern is composed of one class (RE), 5 methods, and 760 lines of source, and therefore does not reflect the concept of a cross-cutting. However, for this case study we choose to include it in order to show how the nature and composition of the concern makes a difference in the results of the analyses.

In order to obtain results for our case study, we instrumented Jakarta `regex` with the instrumentation tool named Clover [10]. Clover measures coverage at three points namely for, methods, statements, and conditions. We ran Clover using 174 test cases that were part of the `regex` package.

2.1. Space Savings

The first question we investigated was the space savings obtained by instrumenting a concern as opposed to the whole program. To answer this question, we instrumented the entire program and then utilized compiler directives within Clover to only instrument the concern of interest. Due to a proof of concept approach, the compiler directives were inserted by hand.

Instrumenting only the concern of interest provides a significant savings in space. For all three concerns, there is approximately 50% savings in space. The results are shown in the second to last row of Table 1. A space reduction is not surprising; however, the results provide insight into the potential amount of savings that can be obtained through instrumenting small portions of code. By instrumenting a smaller portion of code, potentially we will also see a runtime savings as well, due to the fact that there is less instrumentation code to execute. However, the actual runtime savings depends on how frequently the instrumented portions of the code are executed.

2.2. Test suite size

The second question involves determining the test cases in the test suite that execute any portion of code in the concern of interest. In order to select the set of test cases which exercises the concern of interest, we utilized the following

methodology. First, we instrumented the code using Clover. Then, we executed each test case in the test suite, obtaining one coverage report per test case. This isolates the coverage per test case. A typical coverage analysis accumulates the test coverage information as each test case is executed. Next, for each coverage report, we identified the statements in the program, which were part of the concern, and were executed by at least one test case. Each generated coverage report is an XML file of the source code, with tags representing the lines of code executed. This format allowed us to use `grep` in order to identify executed lines of code that were part of the concern. The result of this analysis is a set of all the test cases that cover a portion of the concern. Test cases that are not included in this set do not cover any part of the concern.

Through utilizing this approach, we were able to select only those test cases in a test suite that are associated with a concern of interest. Given that there are 174 test cases associated with `regex` package, there may be instances when we only want to execute the test cases associated with a given concern instead of executing all the test cases in the test suite. While this subset of test cases can be viewed as the set of test cases associated with a use case, the original set of test cases were not organized in such a manner, that is, a manner in which each test case was described with an intended use. In fact, there was no real organization to these test cases besides an enumerated list consisting of input and expected output.

The results of question two are shown in the last row of Table 1. For the clustering concern, we could eliminate 58% of the test suite because these test cases did not result in executing any code in the concern of interest. For the error handling concern, we were able to eliminate 82% of the test cases. The matching concern did not have the same optimistic results. We were unable to eliminate any test cases from the test suite because every test case intersected lines of code that were associated with the matching concern. This was due to the central role of this concern within the `regex` package. Its functionality is needed for the entire package.

These results suggest that it is possible to utilize a concern of interest to help prioritize a set of test cases according to a particular aspect of code. While we are not suggest-

	Methods	Statements	Conditions
Whole Program	55	63	58
Clustering	100	100	–
Error Handling	54	34	25
Matching	100	72	57

Table 2. Percent coverage of concern versus system scope.

ing that the additional test cases be eliminated from the test suite, we are suggesting that there are times when a limited number of test cases could be executed that are specific to a certain concern, thus reducing the time to execute a test suite. Another use of these results is to evaluate a test suite with respect to a concern in terms of the percentage of test cases associated with the concern relative to the size of the concern in lines of code. Depending on the nature of the concern, we may want to allocate more resources in terms of testing such a concern. In particular, if a concern is composed of a large percentage of the overall code in a program, yet there is only a small number of test cases associated with the concern, we may want to construct more test cases.

2.3. Concern-based Coverage

The third question examines coverage results with respect to an individual concern of interest as opposed to coverage at the unit or system scope. These results allow us to further evaluate a test suite according to a concern in terms of coverage. In order to calculate these results, we again utilized Clover and used its coverage reporting feature at the system and class scope in order to measure coverage at the following points: methods, statements, and conditions. We then used the compiler directives to instrument only over the concern of interest.

Table 2 compares the coverage obtained at the system scope (coverage of the entire `regexp` package) with coverage for each concern. The table provides a concern-based perspective of coverage for each concern opposed to a global perspective of the `regexp` package. For instance, method coverage for the system is 55%, however for both the clustering and matching concern 100% method coverage is achieved. Each method participating in the concern has been executed. In addition, we obtain 100% statement coverage for the clustering concern whereas for the whole program, we only obtain 63% statement coverage. Finally, measuring condition coverage, we see that the clustering concern has no conditional statements (represented by the dash in the conditions column), and the error handling concern covered a significantly smaller percentage of conditions than the whole program.

Overall, this table illustrates coverage from a different perspective. We are able to better understand how well each concern has been covered in comparison to the whole pro-

gram. In the case of the clustering concern, we can be confident that the portion of code composed of the concern has been executed, where as for the error handling concern, we may want to add additional test cases to the test suite to exercise the methods, statements, and conditions that have not been covered.

Tables 3 and 4 illustrate percent coverage for each class participating in a concern versus percent coverage for methods, statements and conditions in the concern. This is reported as `classnametotal` versus `classnameconcern` in each table. Table 3 shows results for the two classes that make up the clustering concern. We determined that 100% statement coverage is obtained for the statements that make up the clustering concern, which is higher than statement coverage achieved at the class scope, for both classes. This information indicates that while a class may not be fully covered, a software maintainer can be assured that a concern is fully covered by this test suite based on the coverage measures.

In Table 4, we show similar results. The classes `ReadCharacterIterator` and `StreamCharacterIterator` are not reported in the table because they both reported 0% coverage for all coverage measures. In the case of `RECompiler`, higher coverage is determined at the class scope than concern scope for method, statement, and conditional. Thus, it is possible based on coverage information at the class scope to come to incorrect conclusions about the extent to which a particular concern is being exercised during testing. In the case of the error handling concern, there are several error conditions that are never executed.

Due to the crosscutting nature of concerns, we have shown that coverage associated with encapsulated units such as classes may not always give accurate results. By reporting coverage in terms of a concern, we are able to obtain a more accurate picture of how effective our test cases are for a particular concern. The next section shows how alternatives to and hybrids of statement, method, and conditional coverage can be enabled in a scalable way with respect to concerns.

3. Concern Representation and its Use

Source code is essential for understanding how a concern is scattered throughout the system. However, source code

	Methods	Statements	Conditions
RE _{total}	79	73	57
RE _{concern}	100	100	–
RECompiler _{total}	95	85	80
RECompiler _{concern}	100	100	–

Table 3. Percent coverage of clustering concern versus class scope.

	Methods	Statements	Conditions
RE _{total}	79	73	57
RE _{concern}	100	0	–
RECompiler _{total}	95	85	80
RECompiler _{concern}	67	70	27
RESyntax _{total}	100	100	–
RESyntax _{concern}	100	100	–

Table 4. Percent coverage of the error handling concern versus class scope.

intensive representations are complex in terms of reasoning and analysis. A model of a concern that is more abstract and provides more details about how the different regions of code interact may be a better alternative for modeling concerns than source code.

Robillard and Murphy developed a tool called FEAT, which supports iteratively creating, visualizing and analyzing concerns for Java programs [27, 14]. Recognizing the advantages of FEAT’s concern representation (the concern graph), we decided to explore how this representation could be utilized to guide instrumentation for testing. In particular, we examined the potential use for selective and hybrid instrumentation, where different parts of the concern are instrumented in different ways, to provide different kinds of coverage in different parts of the concern.

3.1. The Concern Graph

With the goal of allowing a software developer to navigate and manipulate a concern representation at a more abstract level than source code, Robillard and Murphy developed the *concern graph* representation, which is a representation for modeling the key structure of a concern, abstracting away the implementation details [27]. While their motivation for constructing the concern graph is entirely different than ours, we believe that once the concern graph has been constructed for the purpose of concern location, it would then be beneficial to continue to utilize the same structure for testing purposes instead of constructing a new representation. The overall approach of concern-based testing could indeed utilize a different kind of graphical representation if desired.

A concern graph explicitly represents key relationships of a concern in order to provide information about how ele-

ments of a concern interact. Only those relationships necessary for understanding a concern are modeled. For example, semantic relationships such as method calls, field uses, and field writes are represented.

As defined by Robillard and Murphy [27], a concern graph is a compact representation of a program that is composed of vertices, where each vertex represents either a global class (class vertex), field member of a class (field vertex), or method member of a class (method vertex), and edges where an edge can be one of six types, depending on the types of vertices it connects: (M,M), (M,F), (M,C), (C,C), (C,M), and (C,F). In addition, each edge is labeled with the semantic information it represents, as follows:

(*calls, m1, m2*): The body of method *m1* contains a call to method *m2*.

(*reads, m, f*): The body of method *m* contains an instruction that reads field *f*.

(*writes, m, f*): The body of method *m* contains an instruction that writes to field *f*.

(*checks, m, c*): The body of method *m* checks or casts an object of type *c*.

(*creates, m, c*): The body of method *m* creates an object of type *c*.

(*declares, c, {f | m}*): Class *c* declares field *f* or method *m*.

(*superclass, c1, c2*): Class *c2* is the superclass of *c1*.

The set of vertices of a concern graph is partitioned into two sets, *part-of* vertices and *all-of* vertices. A vertex *v* is in the *part-of* set if some, but not all of, the program elements represented by *v* implement the concern, whereas a

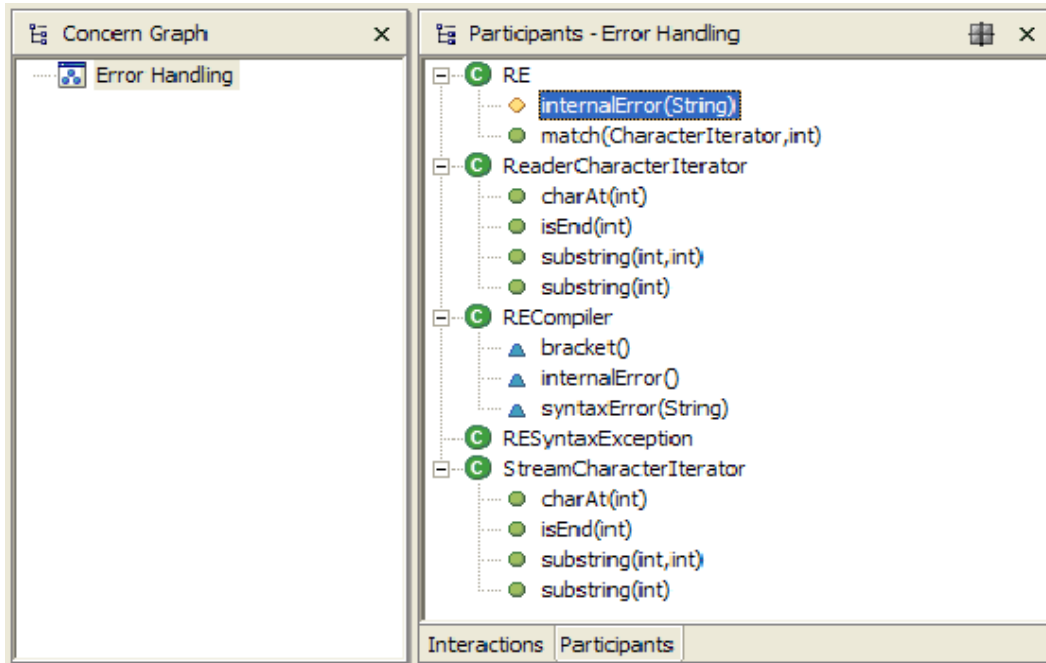


Figure 1. FEAT's concern graph representation of the error handling concern.

vertex v is in the *all-of* set if the program elements represented by v are entirely devoted to the implementation of the concern. For example, if all of the code in a method m is used to implement a concern, then the vertex for method m is constructed as an *all-of* vertex. However, if only some of the statements of method m are used to implement the behavior of the concern, then a *part-of* vertex for method m is constructed.

3.2. Example Concern Graph in FEAT

Using FEAT, we constructed the concern graph in Figure 1 for the *Error Handling* concern of our case study presented in the previous section. We utilized FEAT version 2.1.8, which is implemented as a plug-in for the Eclipse IDE[12]. The figure illustrating the concern graph is only a portion of the information provided by FEAT.

The concern graph in Figure 1 illustrates five classes that participate in the concern. A class without any sub-nodes represents an *all-of* class (e.g., `RESyntaxException`). The other four classes are *part-of* classes, in which only a subset of the methods in the class are *part-of* the concern. In class `RE`, method `internalError` is an *all-of* method and method `match` is a *part-of* method. This fact is not represented pictorially in the figure; however, FEAT provides this information through the highlighting of source code when a user clicks on a method in the concern graph. If the entire method's source is highlighted, the method is

an *all-of* method, while if only part-of the method is highlighted, it is a *part-of* method.

Additionally, in class `RE`, method `match` calls method `internalError`, therefore a *calling* relation edge exists from `match` to `internalError`. In FEAT, this relationship is illustrated in the relations window (This window is not shown in the figure due to space limitations). Class `ReadCharacterIterator` and `StreamCharacterIterator` both have four methods that are marked *part-of*; coincidentally, both classes have methods with the same names, `charAt`, `isEnd`, and `substring`. `RECompiler` has a method named `bracket`, which is designated as *part-of*, which calls methods `internalError` and `syntaxError`, which are both designated as *all-of*. In this example, there are no field vertices.

3.3. Framework for Selective and Hybrid Instrumentation

By utilizing a concern in this form, we have an abstraction of the concern representing only the relevant information of the concern's implementation. The implementation of the concern graph within the FEAT tool maps this representation back to source code. This is important because in order to utilize this representation, we must know which *part-of* a method is associated with a concern. In addition, object-oriented features are also taken into account in the

Algorithm: Concern-based_Instrumentation(C, A, I)

Input: Complete Concern Graph *C*
 Application program *A* containing *C*
 Instrumentation Strategy *I*
Output: Application *A* instrumented for concern *C* and strategy *I*

```

AI = I.all-of_strategy(); //strategy for all-of nodes
PI = I.part-of_strategy(); //strategy for part-of nodes

//Determine scopes for different instrumentation point strategies
//concern instrumentation
foreach method M ∈ C do
  if M.all-of() then
    M.select_instrumentation(AI);
  else // part-of method vertex
    M.select_instrumentation(PI);
// non-concern instrumentation
if (I.instrument_other()) then
  foreach class c ∉ C do
    scope = I.instrument_other_scope();
    scheme = I.instrument_other_strategy();
    foreach method M ∈ scope do
      M.select_instrumentation(scheme)
end Concern-based_Instrumentation

// based on the strategy instrument
// at desired points for different scopes
Method::select_instrumentation(Strategy scheme)

case scheme of
  same:
    foreach point P ∈ same.ipoints do
      this.instrument(P);
  size:
    ipoints = size.range_table[this.howbig()];
    foreach point P ∈ ipoints do
      this.instrument(P);
  weighted:
    value = weighted.w0 * this.num_inedges()
      + weighted.w1 * this.num_callededges()
      + weighted.w2 * this.num_readedges()
      + weighted.w3 * this.num_writeedges();
    ipoints = weighted.range_table[value];
    foreach points P ∈ ipoints do
      this.instrument(P);
end case
end select_instrument

```

Figure 2. Selective and hybrid instrumentation using a concern graph.

representation, including polymorphism in the calling relationships, field read and write operations, and inheritance.

In our work, we investigated how the structure of the concern graph can guide selective and hybrid instrumentation. By selective we mean choosing whether to instrument different scopes, or different parts-of a method, and by hybrid we mean different coverage measures can be utilized at different scopes. Such instrumentation provides more specific information in terms of coverage over a concern, as well as we can instrument parts of a concern in more detail. For example, it may be too expensive in terms of time and space to instrument an entire program or even a whole concern in order to obtain def-use information; however, it may be practical to instrument only a portion of a concern, or only certain fields in such a manner. Therefore, by utilizing a concern graph to guide instrumentation we may be able to obtain more detailed coverage for selected program entities.

Figure 2 shows an algorithm for implementing a parameterized framework for selective instrumentation for a program based on a concern graph. The framework is parameterized to allow for any of a set of coverage measures to be used for a given scope. Possible coverage measures could be methods, statements, conditionals, or def-use, given the concern graph representation. The framework is also parameterized such that the scope for a given coverage measure is flexible.

The selective and hybrid instrumentation algorithm for concern-based testing takes a program, concern graph, and instrumentation strategy as input. The algorithm produces an instrumented program according to the desired strategy. The algorithm expects a complete concern graph, that contains all field, class, and all-of or part-of method nodes, where all-of class nodes have been automatically expanded to create the corresponding all-of methods. If only the concern is to be instrumented, then only the source code for the classes containing the concern code needs to be made available. The instrumentation strategy indicates which coverage measure should be implemented for each all-of and part-of node in the concern graph as well as a coverage measure for the remainder of the program, if the non-concern part of the program is to be instrumented at all.

The function `select_instrumentation` is invoked for each method in a scope to be instrumented according to a given instrumentation strategy. The framework provides for three general instrumentation strategies to be used for instrumenting a given scope. The *same* strategy instruments all methods according to the same coverage measure. The *ipoints* field of *same* indicates the instrumentation points of the method based on the measure desired. The *size* strategy instruments a given method based on the range in which its size falls. The size of the method (*howbig*) could be defined in terms of the number of lines of code, or the number of outgoing edges from its node in the concern graph.

A table indicates the corresponding coverage measure for different ranges of method sizes. Therefore, different coverage measures can be utilized for different methods based on the method size. The table allows for easy definition and modification of coverage measure for different method size ranges.

The *weighted* strategy is the most general in that it allows for the coverage measure to be based on a number of factors, including the number of incoming, call, read, and write edges involving the method's node in the concern graph. In addition, the different factors can be weighted as desired. A table is used to identify the desired coverage measure for different values of the weighted sum. In fact, the *same* and *size* strategies are subcases of the *weighted* strategy, but we separate them for simplicity and clarity. With these strategies, a number of different heuristics can be implemented by altering either the tables or weights or the overall instrumentation strategy for all-of or part-of nodes of the concern graph, and the rest of the program.

The algorithm runs in time proportional to the number of points to be instrumented in the program, which is at most linear in the size of the program. If only the concern is to be instrumented, then it is at most linear in the size of the concern. Note that if def-use coverage is desired, then more analysis is needed to compute the coverage points.

4. Related Work

4.1. Priority-based and Regression Testing

Priority-based testing typically tries to prioritize a test suite according to some testing objective. For example, one objective is rate of fault detection, that is, how quickly are faults detected based on the ordering of test cases. Another objective is total function coverage prioritization, which orders test cases according to the number of functions covered by a test case. Several different prioritization objectives have been studied [30, 35, 2].

While our work extracts a subset of test cases for testing, we are not suggesting that this subset of test cases will improve fault detection. We are suggesting that a subset of test cases can be extracted to help guide a software maintainer in making changes and helping the maintainer initially test the assigned crosscutting region of code. Existing regression testing techniques [29, 17] can then be utilized to determine the overall reaching effects of the modified code.

Regression testing is the process of validating modified software to provide confidence that the changed portion of code has not adversely affected the rest of the program. Regression test selection techniques select a subset of the test suite in order to reduce the cost of testing [28, 17, 3, 33, 22, 4, 31, 20, 29]. Rerunning all the test cases can be expensive. Many regression test selection

techniques are based on modification to the code. In contrast, test selection in concern-based testing is not based on changes to the code, in fact, changes to the code may not have been performed yet. Instead, testing is based on an arbitrary region representing a concern relating to the maintenance task, a region that most likely spans the developer's modularization boundaries.

4.2. Concern Location utilizing Dynamic Slicing and Coverage

The problem of locating features or concerns within a large program is not a new problem. Many tools exist to help a maintainer locate code, such as grep [1], code browsers [23, 15, 16, 25, 18, 27, 21], cross-reference databases [5], program slicers [32, 34], and feature exploration tools [27]. In addition, concept analysis has been used to locate features in source code [13].

One approach to locating program features is based on utilizing dynamic execution slices [34]. In order to determine the source code associated with a particular concern of interest, the approach is based on constructing two new sets of test cases. The first set is the invoking set, which is constructed based on the notion that this set of test cases will execute code associated with the concern of interest. The second set, the excluding set, is constructed to contain test cases which when run do not cover the concern of interest. Utilizing these two new test sets, a software maintainer would then use a coverage tool and execute the invoking and excluding test sets. In order to determine the code that is associated with the specified feature, the $\bigcup_{invoking} - \bigcup_{excluding}$ is calculated, which gives code covered by the invoking test set and not the excluding test set. While this is potentially a technique for identifying the source code, the maintainer needs to construct a new set of test cases, which depending on their quality may or may not be helpful. The test cases may execute more code than is actually associated with the concern of interest, or certain exceptional parts of the concern may not be executed at all. The software maintainer does not rely on an existing set of test cases associated with code. In contrast, this paper exploits code identification techniques for concerns and determines the set of test cases that are associated with a concern based on an existing test suite.

4.3. Aspect-Oriented Programming

Object technologies face difficulties when deriving solutions to problems that crosscut several programming concerns. Difficulties arise due to the inability to clearly separate certain programming behaviors into distinct units, therefore certain behaviors are programmed across multiple units causing code to be difficult to understand and reuse.

Aspect-oriented programming (AOP) tries to solve some of these problems by providing mechanisms that allow a programmer to separate concerns (properties or features of interest), and then allow the underlying AOP environment to weave the concerns together into one program.

Ongoing research pertaining to many different facets of aspect-oriented programming is being explored. In particular, the design and implementation of aspect-oriented programming languages [19, 24], the design of aspect-oriented programming environments [7, 6, 8], aspect-oriented software processes [9], and techniques for finding and defining concerns in a program [15, 27] are all growing areas of research.

Aspect-oriented research most closely related to this work pertains to environments for supporting the browsing of crosscutting concerns within a program [23, 15, 16, 25, 18, 27, 21]. Different types of relationships between scattered parts of the code-based are modeled at the source level by utilizing a query-language to extract useful information from the code base. Existing tools include HyperJ [23], Aspect Browser [15], Aspect Mining tool [16], JQuery [18], an extension of the earlier tool QJBrowser [25], and FEAT [27, 21]. These code locating techniques complement our work.

5. Conclusions and Future Work

The main contribution of this paper is a demonstration of the possible savings in testing overhead and the increased precision in coverage information that can be obtained for a software maintainer if testing tasks are performed with respect to the scattered regions of code that have been assigned for maintenance. Based on these results, we propose that test cases could be organized with respect to concerns of interest for maintenance, and prioritized such that a software maintainer assigned a set of maintenance tasks for a given concern, or feature, could reduce the instrumentation space requirements, reduce the test suite execution time, and obtain more precise coverage information with respect to the arbitrary region of code included in their maintenance concern. In addition, we have presented a parameterized framework for selective instrumentation for a program based on a concern graph. The framework allows for different instrumentation strategies to be implemented in different scopes of the program, based on the nodes and edges in the concern graph. This concern-based approach to testing would enable more scalable testing of large programs during maintenance, but still achieve testing beyond unit testing.

We are currently searching for larger case studies, with available test suites to examine the effects of this approach on larger programs. In addition, we are examining different kinds of coverage within a concern, based on object manip-

ulations, and providing information about the interactions of the concern and the rest of the program through escape analysis information.

References

- [1] A. Aho. Pattern matching in strings. In *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, 1980.
- [2] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, Sept. 1995.
- [3] T. Ball. On the limit of control flow analysis for regression test selection. In *International Symposium on Software Testing and Analysis*, pages 134–142, 1998.
- [4] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*, 1994.
- [5] Y. F. Chen, M. Y. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.
- [6] M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2000.
- [7] M. C. Chu-Carroll and S. Sprenkle. Software configuration management as a mechanism for multidimensional separation of concerns. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE)*, 2000.
- [8] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine-grained software configuration management. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2002.
- [9] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. The dimension of separating requirements concerns for the duration of the development lifecycle. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA)*, 1999.
- [10] Clover: Code coverage tool for Java. 2002. <<http://www.thecortex.net/clover/>>, (March 1, 2003).
- [11] CodeSurfer Home Page. January 2001. <<http://www.codesurfer.com>>. (March 1, 2003).
- [12] Eclipse. 2001. <<http://www.eclipse.org/>>, (March 1, 2003).
- [13] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, Mar. 2003.
- [14] FEAT Eclipse Plug-in. 2002. <<http://www.cs.ubc.ca/mrobilla/feat2/>>, (March 1, 2003).
- [15] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-dimensional Separation of Concerns in Object-oriented Systems*, 1999.
- [16] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering*, 2001.
- [17] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2001.

- [18] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *3rd International Conference on Aspect-Oriented Software Development*, 2003.
- [19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.
- [20] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–40, Jan. 1996.
- [21] A. Lai and G. Murphy. The structure of features in Java code: An exploratory investigation. In *First Workshop on Multi-dimensional Separation of Concerns in Object-oriented Systems at OOPSLA*, 1999.
- [22] H. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings on the Conference on Software Maintenance*, 1991.
- [23] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson Research Center, 1999.
- [24] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. Technical report, IBM T. J. Watson Research Center, 2000.
- [25] R. Rajagopalan and K. D. Volder. QJBrowser: A query-based browser for exploring crosscutting concerns in code. Technical report, University of British Columbia, 2002.
- [26] Java Regular Expression package. 2001. <<http://jakarta.apache.org/regexp/>>, (March 1, 2003).
- [27] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*, 2002.
- [28] G. Rothermel and M. Harrold. A safe, efficient regression test set selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.
- [29] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++. *Journal of Software Testing, Verification, and Reliability*, 10(6):77–109, June 2000.
- [30] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.
- [31] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Proceedings of the International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, 1997.
- [32] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [33] L. J. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. In *Software Quality Week*, 1997.
- [34] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *IEEE Application Specific Software Engineering and Technology*, 1999.
- [35] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.