

Aspect Language Features for Concern Coverage Profiling

Hriday Rajan

Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296

hr2j@cs.virginia.edu

Kevin Sullivan

Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2206

sullivan@cs.virginia.edu

ABSTRACT

In program profiling to assess test set adequacy, a challenge is to select code to be included in the assessment. Current mechanisms are coarse-grained; biased to dominant modularizations; require tedious, error-prone manual selection; and leave tester intent implicit in inputs to testing tools. Aspect-oriented constructs promise to improving testing in two ways: by improving our ability to select the code to include in adequacy criteria, and by documenting selection intentions in declarative form in the code itself. One problem is that current join point models do not reveal program behavior in enough detail to support white-box coverage analysis. Our contribution is the formulation, prototyping, and evaluation of a language-and-tool-based approach to white-box coverage adequacy analysis that we call *concern coverage*. We develop and evaluate one instance of the general idea in which branches, in particular, are exposed as join points to support branch coverage analysis of crosscutting concerns. Our results are consistent with the claim that the idea has the potential to improve test coverage analysis.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Testing tools – *code coverage analysis, testing*. D.3.3 [Programming Languages]: Language constructs and features – *generalized join point model, generalized advice*.

General Terms

Measurement, Design, Experimentation, Languages, Verification.

Keywords

Concern coverage, generalized join point models, generalized advice, Eos, C#, coverage adequacy criteria.

1. INTRODUCTION

Exhaustive testing of computer programs is usually infeasible [6][24], and partial testing set can only show the presence of faults, not their absence [13]. Nevertheless, partial but systematic

testing is a widely used alternative that often reveals faults and improves confidence in program behavior. A *test adequacy criterion* is a predicate defining what parts of a program must be exercised to constitute an adequately systematic test [24][29].

The problem that we address in this paper is that the state of the art in selecting the code to be included in an adequacy criterion falls short. First, current mechanisms are too coarse-grained and too biased to traditional modules to support fine-grained selection of scattered code implementing crosscutting concerns. Second, current mechanisms require tedious, error-prone manual selection of included elements, usually through the graphical user interface (GUI) of a testing tool. Commercial tools for coverage analysis such as IBM Rational's Pure Coverage [49], and open source tools such as Quilt [64], provide control over inclusion of code at the file, class, and function level only, and only by manual enumeration through a GUI. Third, expressing adequacy criteria by selecting modules alone does not clearly express the intent of the tester, potentially complicating test evolution.

To address these problems we present a new approach that we call *concern coverage*. The idea is to express test adequacy criteria in abstract, declarative form within the code to be tested, in the form of aspect-oriented [34] pointcut constructs in the style of AspectJ [3]. The programming environment automates the instrumentation of selected behaviors for coverage analysis, and the collection and presentation of test results. This approach relieve testers of having to select scattered code manually and it not only supports the abstract expression of crosscutting adequacy criteria, but does so within the primary development artifact in most cases: the code. A key issue that we address is that current join point models do not generally expose all of the behaviors needed to enable white-box coverage analysis.

The rest of this paper is organized as follows. Section 2 presents background on coverage analysis and analyzes problems with the current state of the art. Section 3 presents our approach and the language extensions needed to support it. Section 4 describes our framework for implementing tools based on these extensions. Section 5 describes *AspectCov*, a coverage analysis tool built using our framework, to support an initial evaluation of our ideas. Section 6 discusses related work. Section 7 concludes.

2. MOTIVATION

Code coverage analysis is a technique to determine whether a set of test cases satisfy an adequacy criterion. It helps find areas of a program not sufficiently exercised by a set of test cases, gives a measure of the quality of a test set, and can determine whether a test set meets coverage requirements. In general, adequacy is

assessed against selected parts of a code base, and those parts are scattered across the program. Measuring coverage requires that the source code be instrumented at points of interest. Something like a break point in debugging has to be inserted at each point of interest. To instrument code manually is a burden on the tester.

Code coverage tools are meant to help remove this burden by automating the task of code coverage analysis. Coverage tools are designed with specific adequacy criteria in mind. For example, a tool might allow statement coverage, path coverage, condition coverage, segment coverage etc. These tools, however, limit the flexibility of code coverage. The limited set of code coverage criteria is all a tester can choose from, and the code against which adequacy is to be evaluated can be designated only by the limited means supported by the given tool.

In practice, testers are often not interested in coverage of all program elements, but rather in selected elements. One might wish to select only non-library and non-environment-generated code, for example. Similarly, a tester of one module might want to include the code of the module or function that is being tested, and perhaps closely related code elsewhere in the system, but not more distantly related code. In her proposed roadmap for testing [27], Harrold points out the need to express selective test adequacy criterion in the context of component based systems.

We need to identify the types of testing information about a component that a component user needs for testing applications that use the component. For example, a developer may want to measure coverage of the parts of the component that her application uses. To do this, the component must be able to react to inputs provided by the application, and record the coverage provided by those inputs. For another example, a component user may want to test only the integration of the component with her application. To do this, the component user must be able to identify couplings between her application and the component. (pp. 3)

She also outlines the need for expressive, and even code-based, representations of testing policies in general and coverage criteria in particular:

We need to develop techniques for representing and computing the types of testing information that a component user needs.... Likewise, standards for representing testing information about a component, along with efficient techniques for computing and storing this information, could be developed. For example, coverage information for use in code-based testing or coupling information for use in integration testing could be stored with the component. (pp. 3)

More generally, a tester might wish to define test adequacy criteria in terms of coverage of code that relates to concerns that are lexically spread across the code base. For example, one might want to cover scattered synchronization, transaction, serialization, or security related code. Schulte [54] has indicated that there is a need to represent such selective and crosscutting adequacy criteria in large projects at Microsoft, where testers are often interested in testing selected, scattered parts of large code bases.

We see two basic kinds of mechanisms for representing adequacy criteria: tool-centric and language-centric. In the former approach,

common in commercial tools, the coverage tool provides a fixed set of adequacy criteria, with the possibility of manually selecting code elements against which adequacy is to be evaluated, limiting the tester to the fixed set. The latter approach provides a language front end to the analysis tool. The flexibility in representing adequacy criteria depends on the expressiveness of this front-end language. Our approach is of the latter kind.

3. OUR APPROACH

Our work views an adequacy criterion as a crosscutting concern in the sense that it is a requirement for the testing process that maps to monitoring execution of program elements scattered across the code base. Existing mechanisms for performing this monitoring operate at traditional level of modularity and thus can suffer from what has been called the tyranny of dominant decomposition [58]. In a nutshell, they make it hard to select scattered code elements that implement crosscutting concerns.

Assessing the satisfaction of test adequacy criteria requires monitoring of key points in the execution of programs under test. In aspect-oriented terminology, interesting points in execution are called join points. An adequacy criterion, like a pointcut, selects a subset of join points. The idea behind our approach is to expose key points of interest as join points, and to express the code to include in adequacy criteria using declarative pointcut constructs. For example, one could express the criterion *Coverage of all non-recursive external function calls on types in the package foo.bar and all its direct and indirect sub-packages* as in the following aspect, and then have a tool automate the coverage analysis. The type pattern *foo.bar..** (in AspectJ [3]), represents all types in the package *foo.bar* and all its direct and indirect sub-packages. In the later sections, we will complete this aspect body.

```
aspect FunCov{
    // Monitor coverage of all external function calls on types in foo.bar..*
}
```

These *test coverage aspects*, included in the code base, are meant to support automated, fine-grained, selective instrumentation for required profiling and data collection. They address the three problems. First, they support the expression of crosscutting coverage criteria. For example, *FunCov* selects calls in multiple packages and sub-packages using a simple type pattern *foo.bar..**.

Second, they relieve the tester of the task of manually selecting and reselecting files, packages, and functions. For example, to select the set of program execution points constituting the criterion represented by *FunCov* using a GUI based tool, a tester would have to go through all files and select all calls to any method in the package *foo.bar* and all its direct and indirect sub-packages, and then manually determine whether given calls are non-recursive and external.

Third, these coverage aspects document and communicate tester intent in explicit, precise, abstract, and machine-readable forms. Such expressions can also remain valid as the source code evolves. It would be easy to add a new call to a method in *foo.bar* package to a program and forget to update the test set and the adequacy criterion. Using predicates to express the code to be covered resolves this issue.

Unfortunately, current aspect language designs in the AspectJ style are not ideally suited to our task. In the rest of this section, we discuss the need for the language model extension to support our ideas and describe how the extensions incorporated in our language, *Eos-T* (*T for testing*), achieve some of our goals. We then describe our framework and its embodiment in *AspectCov*.

3.1 Need for Language Model Extension

The promise of aspect-orientation for modularizing crosscutting concerns encouraged us to use it to represent adequacy criteria in a modular way. In particular, pointcuts appeared to be a promising mechanism to select code elements that bear on the satisfaction of adequacy criteria. For example, the *FunCov* criterion discussed earlier could be represented by the simple pointcut in this code.

```
aspect FunCov{
  /* Apply code coverage to all the join points selected by */
  call (* foo.bar..*.(..) && ! within(foo.bar.*);
}
```

Here, the tester uses the pointcut `call (* foo.bar..*.(..))` to specify that adequacy is to be measured against all method calls on types in the direct or indirect sub-package of *foo.bar*, and then uses the pointcut `!within(foo.bar.*)` to exclude calls from within the package or sub-packages themselves. All join points in the lexical scope of the specified types are selected by the *within* pointcut designator. The operator `!` complements the set to select all join points that are not in the lexical scope of the specified types.

As it turns out, the join point models and pointcut languages of current aspect languages are not fine grained enough to express adequacy criteria at a granularity needed for white-box testing. For example, no aspect languages that we know of, except for versions of our own Eos language, expose branches as join points, yet branch coverage is a major variety of white-box testing. This shortcoming precludes precise, abstract expression of crosscutting *branch coverage* criteria. In general, aspect-oriented coverage analysis is limited by the join point model of a given language. AspectJ, the most mature and well-developed model, recognizes method call & execution, field get and set, exceptions, control flow, etc., as join points but not basic blocks or branches, statements, reads and writes of all variables, etc.

The problem reflects basic and well justified language design decisions. Ossher and Tarr [47] argue against statements as join points, for example, questioning their utility, and expressing concern about the stability of references to statements-level join points in the face of program evolution. AspectJ [3] similarly adopted a join point model limited mainly although not entirely to stable interface elements (e.g., calls to named methods). One of the main contributions of this work is the idea that aspect-oriented profiling, with applications to coverage analysis, provides what is perhaps the first compelling example showing the potential utility of finer-grained, *white-box join point and pointcut models*. As an initial test of utility, we present a language that exposes branches, in particular, as join points, and a system for branch coverage analysis of crosscutting concerns. We do not claim to have worked out all of the language design issues or other implications of providing such support in industrial-strength language designs.

In another dimension, most aspect languages support one primary action at the join points selected by aspects: namely weaving of advice to modify the base program behavior. This mechanism can support the profiling needed for coverage analysis, but it is not an

efficient mechanism for this purpose. Other mechanisms, such as those of Ball and Larus [4][5], have superior performance in many cases. Apart from weaving advice and coverage analysis, one might want to schedule such points for manual inspection, monitor the bugs reported in the program segments written by a particular programmer, automatically generate/update related sections of the program documentation, etc. The potential action list is not limited to the actions mentioned above but might include any other activity in the software development life cycle amenable to automation. We propose a decoupling of pointcut expressions from the actions to be taken at designated join points, with advising as a special case.

We thus introduce the idea of *generalized advice*: directives that are associated with pointcuts within aspects and that are processed by supporting tools to cause actions other than weaving of advice code at the designated join points. While such actions occur at compile time, they can be relevant to tasks across the life-cycle, e.g., document generation, inspection, etc. In this work we present and evaluate one instance of this idea for test coverage profiling.

To evaluate the feasibility and potential utility of our ideas, we have designed and implemented Eos-T, a version of the Eos aspect-oriented language [18][50] with added support for generalized join point and pointcut models and generalized advice. Eos, an aspect-oriented version of C# [43] [44], was the first aspect language to support both first-class aspect instances and instance-level advising along with rich join point and pointcut models in the AspectJ style. Eos-T is implemented in a fully functional compiler, comprising approximately 40,000 lines of code (most inherited from Eos). With Eos-T, we have been able to demonstrate the feasibility and potential utility of selective code coverage analysis by compiling large, open source C# programs as exemplars. Eos-T is available to the research community. In the next subsection, we will discuss the language extensions to Eos to enable the representation of test adequacy criteria as aspects.

3.2 Eos-T and its Extended Join Point Model

To begin, Eos-T extends the join point model of Eos to include basic blocks as join points. Such join points enable selective branch coverage profiling based on declarative pointcut expressions. The Eos-T join point model includes statement-level join points, iterations, and conditionals. These statements, called predicates in the control flow graph (CFG) terminology [1], along with method calls and returns, mark the beginnings and ends of basic blocks.

A conditional statement in C# is an *if* statement or a *switch* statement, and an iteration statement is one of *while*, *do-while*, *for*, or *foreach*. The Eos-T pointcut language allows one to select such join points using the *conditional* and *iteration* pointcut designators (PCDs). The syntax of these PCDs is as follows:

```
conditional_pointcut: CONDITIONAL (conditional_construct)
conditional_construct : IF | SWITCH | *
iteration_pointcut: ITERATION (iteration_constructs)
iteration_constructs : iteration_construct |
                    iteration_construct || iteration_construct
iteration_construct: WHILE | FOR | DOWHILE | FOREACH | *
```

The following pointcut expressions show some of the usage of the *conditional* and the *iteration* PCDs.

```
pointcut allConditional() : conditional(*);// Select all conditionals
```

pointcut allif(): conditional(if); // Select all if statements
pointcut allswitch(): conditional(switch); // Select all switches
pointcut allIterations(): iteration(); // Selects all iteration statements*
pointcut allwhile(): iteration(while); // Selects all while statements
pointcut allfor(): iteration(for); // Selects all for statements

In addition to these new PCDs, EOS-T supports the PCDs shown in Table 1. The PCDs can be composed together by || (disjunction) and && (conjunction) operators. These PCDs and patterns are mostly inherited from AspectJ, and are discussed in more details in the AspectJ user’s manual [3].

Most of the PCDs described in the table are anchored to interface elements, and select groups of join points with respect to that interface element. For example, the call PCD is anchored to a method name interface element, and it picks out all the calls to the specified method regardless of where the calls originate. The *conditional* and the *iteration* PCDs, which are not anchored to interface elements, select all elements of the given construct type.

To refine the selection of the join points further, these PCDs can be composed with other PCDs like *within*, *withincode* etc.

For example, the following pointcut expression matches all the *if*’s within the *TestHarness* class.

pointcut ifTestHarness() : conditional(if) && within(TestHarness);

Similarly, the following pointcut expression matches all *while* loops in the *Test* function of the class *TestHarness*.

*pointcut whileTest() : iteration(while) &&
 withincode(public void TestHarness.Test());*

The join point model of Eos-T thus allows the expression of more complex adequacy criteria than can be expressed by traditional file, module, and function enumeration. In addition, it relieves the tester of the tedious and error-prone task of manually selecting and reselecting the files, packages, functions that are to be included in the coverage analysis. It doesn’t yet directly support selection of individual branching constructs within procedures.

3.3 Generalized Advice

Most aspect languages and approaches including AspectJ and the original Eos language only support the weaving of advice code at selected join points. The possibility of enabling other actions opens up new possibilities for expressing actions across the software lifecycle as part of the source code itself, in an abstract, *modular* and *composable* form. Representing actions explicitly and abstractly within the code base promises to increase the reliability of certain lifecycle activities, by representing them in the artifacts most directly manipulated by developers, in a form in which they are easier to understand, subject to automation, and maintainable along with the code.

For example, consider the documentation and realization of a test adequacy criterion. Traditionally, one would document it in a test plan and realize it by selecting options in the analysis tool. At the beginning of each test session or after test case execution is finished, depending on the coverage analysis tool being used, this criterion will be replicated in the testing environment by manually selecting the program elements that are to be included in the coverage analysis.

Table 1: Pointcut designators (PCDs) supported by Eos-T

PCD	Explanation
<i>call(Signature)</i>	Matches every methods call site matched by the signature
<i>execution(Signature)</i>	Matches execution of every method matched by the signature
<i>fget(Signature)</i>	Matches variable read for variables matching the signature
<i>fset(Signature)</i>	Matches variable write for variables matching the signature
<i>initialization(Signature)</i>	Matches object initialization/constructor calls matching the signature
<i>handler(TypePattern)</i>	Matches exception handlers catching an exception of type matched by the pattern
<i>adviceexecution()</i>	Matches execution of every advice
<i>within(TypePattern)</i>	Matches each join point within the type matched by the pattern
<i>withincode(Signature)</i>	Matches every join point within the method matched by the signature
<i>pget(Signature)</i>	Matches property read for variables matching the signature
<i>pset(Signature)</i>	Matches property write for variables matching the signature
<i>conditional(if/switch)</i>	Matches conditional join points (if and switch statements)
<i>iteration(while/dowhile/for/foreach)</i>	Matches iteration join points (while, do-while, for and for each statements)

The changes in the adequacy criterion can be introduced either by the management, which will be first reflected in the test plan and then replicated in the testing environment, or by the testing staff, which will first experiment with the criteria in the testing environment and then document it in the test plan.

In both cases, additional maintenance effort is required to make the documentation in the test plan and its realization in the testing environment consistent. This additional maintenance is required because the translation process from test plan documentation to the testing environment (and vice versa) contains a manual component: selecting program elements for analysis. Providing language and tool support to automate this manual component reduces the required additional effort. This support is provided in Eos-T as generalized advice. An instance of generalized advice has the following syntax:

action IDENTIFIER(): Pointcut;

An identifier depicting the name of the action follows the new keyword *action* followed by a pointcut expression. The pointcut expression identifies the subset of program elements at which the action should be taken. This extension enables us to embed useful actions in the source code. These actions are collected and passed to corresponding tools along with the pointcut expressions at which actions should be taken. Tools recognizing these extensions can automatically perform these embedded actions.

The actions that are recognized are independent of the Eos-T language model. The actions that are recognized in a development environment are defined by the tools present in the environment. To add a new action to the list of recognized actions, addition of a new tool that recognizes that action is sufficient. In the next section, we describe our framework for implementing tools that recognize these extensions.

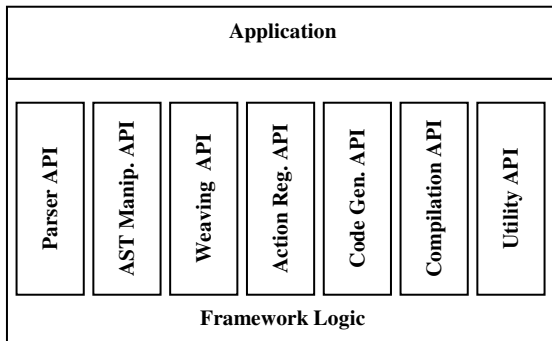


Fig. 1 Framework overview

4. FRAMEWORK

As illustrated in Figure 1, our framework provides application programming interfaces (APIs) for source code parsing¹, abstract syntax tree (AST) manipulation, code weaving, action registration, code generation and code compilation. The utility API provides miscellaneous functions like command line argument processing, database manipulation, string manipulation, etc. The framework logic consists of three stages: initialization, run, and close up. In the initialization stage, the framework parses the source code and constructs the abstract syntax tree. While parsing, it collects all the encountered actions. A tool built on top of our framework uses the action registration interface to register the action it intends to handle with the framework.

After initializing itself, the framework calls a tool specific initialization handler. Similarly, in each of the three stages, the framework iterates through the collected actions and calls appropriate handlers, thus allowing each tool to perform its task. These tools are invoked in the order in which they have registered with the framework.

As opposed to other frameworks for implementing verification tools, e.g., Aristotle [28], our framework supports aspect-oriented constructs such as pointcuts, advice, etc., and offers an interface to perform crosscutting actions. Java byte code transformation tools such as BCEL [8], SOOT [57], etc. similarly allow arbitrary transformation of java classes; however, they do not recognize crosscutting structure and do not allow crosscutting transformation. In the next section, we demonstrate the use of this framework, along with the aspect language extensions embodied in Eos-T, to implement our *concern coverage* method and system.

5. THE ASPECTCOV TESTING TOOL

Our tool for concern coverage analysis is intended to provide the ease of tool-based instrumentation coupled with the flexibility of specifying *concern coverage* criteria using a declarative language. The tester can use the entire pointcut sub-language of Eos-T to

specify the coverage criteria. Use of the declarative language enables explicit, precise, and abstract specification of the tester's intentions.

The ability to select a subset of program elements using the declarative language improves the efficiency of code coverage analysis. Even in scenarios where complete coverage is essential, the ability to select will improve efficiency in successive test measurements. For example, after a test set execution, in subsequent testing user can instruct *AspectCov* to cover and track metrics for only the uncovered portion of the design. In this way, coverage is directed towards the problematic design areas, reducing the overhead in each successive test set execution and speeding the testing process. Using an expressive pointcut language also allows significantly more sophisticated coverage criteria to be represented.

We built our code coverage tool using a combination of our framework and the NUnit unit-testing framework [46]. Our tool performs the following steps to achieve its goal.

1. It registers itself with the language framework to receive compile-time notifications for *AspectCov* actions.
2. In the initialization stage, it collects all the *AspectCov* actions. As discussed previously, every action contains a pointcut expression. In case of *AspectCov* action, this pointcut expression specifies the sub-set of join points that correspond to a given coverage concern.
3. Our tool selects a union² of these sub-sets of join points that need to be instrumented.
4. It then generates and adds infrastructure to serialize the reflective information at the join points and send it to a coverage information collection server running at a well-known port. This communication is performed using a shared stream if the test and analysis machine are the same.
5. The selected join points are instrumented to record the coverage information. Optimal instrumentation algorithms could be use to actually instrument the code.
6. The abstract syntax tree of the instrumented program is compiled into an assembly³.
7. This instrumented assembly is loaded into the NUnit unit-testing framework..

The tester can now execute the test-suite of the program using the familiar graphical user interface (GUI) of NUnit and obtain coverage information for the specified concern. *AspectCov* stores the coverage information as a coverage matrix, the rows of which represent the join points for which coverage is measured, and the columns of which represent the tests in the test suite. If data from previous runs for the project is available, it also presents the metrics, changes across components, and change across tests, as

¹ Currently our parser API only parses C# source code; however, we are extending it to include support for the intermediate language (MSIL).

² More sophisticated mechanisms such as [1], [4], [5], [25], [36], [48], [52], etc. can also be used.

³ Assembly is the Microsoft .NET framework's equivalent of an executable.

described by Elbaum, Gable, and Rothermel [16], to give an idea of the change in coverage between versions.

As can be observed in Figure 2, we have introduced two new GUI elements to NUnit: a button to start the code coverage analysis and a tab to display the analysis results. Currently we use the NUnit testing tool; however, our approach can be extended to other testing infrastructures. Figure 2 shows coverage calculated for the selected concern at the bottom of the right hand panel.

The overheads incurred by *AspectCov* are as follows:

1. Development time overhead: The tester’s time spent to write the coverage aspect. In our experience, this overhead is small compared to that incurred in manual selection of program elements.
2. Compilation overhead: The code base and aspect need to be compiled. Currently our approach works at the source code level so recompiling is necessary. This overhead might be eliminated by working at the MSIL level for C# and byte code level for Java.
3. Run-time overhead: At every join point subject to coverage analysis, additional code is inserted to record the coverage results. This code segment can be further optimized. For other possible applications of our approach, the first two overheads will remain the same. The third overhead will vary depending upon the specific application.

AspectCov currently instruments all join points selected by a given pointcut, but it does not try to further optimize the number of join points instrumented. Optimal profiling approaches such as the Ball et al. approach [4] will be used in future. Finding a spanning set [41] of the coverage concern will further optimize the number of join points instrumented. Nevertheless, we see that instrumentation is added to the application only at selected join points, which results in a decrease in the size of the resulting application and run-time of test execution.

To compute test set adequacy, *AspectCov* first statically determines the set of join points that belong to the coverage concern and then at run-time calculates the subset of this set that is executed by the test-set to determine coverage percentage. Therefore, the adequacy criteria may only contain pointcut expressions that are statically determinable [3]. As with the *declare* constructs in AspectJ, the pointcuts *cflow*, *cflowbelow*, *this*, *target*, *args* and *if* may not be included in the adequacy criteria, directly or indirectly, because they need run-time information to be accurately determined.

5.1 Example coverage scenarios

To use *AspectCov* for code coverage analysis, one simply adds test coverage aspects to the source code base. The code coverage concerns are expressed in the aspects. Multiple code coverage aspects can co-exist in the code base. Code coverage aspects can co-exist with other aspects. Among other things, this approach does allow components to carry their own testing requirements in the form of component-local coverage aspects.

Consider the adequacy criterion *Every modification to the value field of the class Model should be covered*. Most coverage analysis tools do not provide such fine-grained selection.

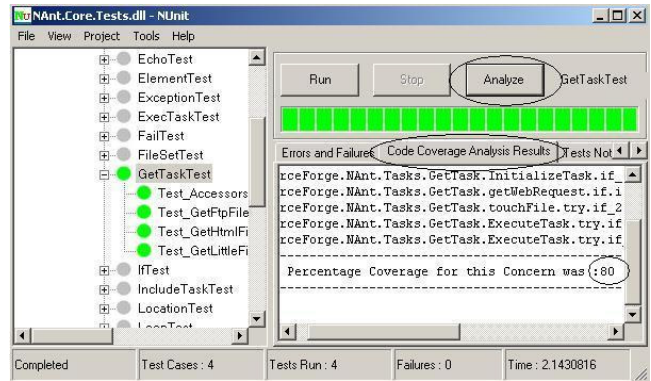


Fig. 2 Screen shot of *AspectCov* integrated with NUnit.

As an approximation one could manually analyze all methods to find methods where the value field of the class Model is being changed and include those in the adequacy criterion. Using our approach, we will represent the criterion as follows:

```
aspect ValueModificationCoverage {
    action AspectCov(): fset(public int Model.value)
}
```

The aspect *ValueModificationCoverage* contains only one *AspectCov* action, which is applied to all program elements selected by the pointcut expression, *fset(public int Model.value)*. The pointcut starts with the PCD *fset*, for field set, which matches all program elements where a field is modified. The pointcut expression further narrows the match by specifying the pattern (*public int Model.value*). This pattern means match only when the field modified is a *public* field of the type *integer*, and it belongs to the class *Model* and has the name *value*.

Now let us reconsider the adequacy criterion discussed, but not concretized, in Section 3. Adding the following simple coverage aspect to the code base will be sufficient for representing the adequacy criterion, whereas using traditional mechanism a manual analysis of each method in the program will be needed to determine if it contains a method call to any type in the package *foo.bar* and all its direct and indirect sub-packages.

```
aspect FunctionCoverage {
    action AspectCov(): call (* foo.bar..*.*(..)) && ! within( foo.bar..*);
}
```

Similarly, the adequacy criterion *error code coverage* and *decision coverage* will be simply represented as:

```
aspect ErrorAspectCoverage {
    action AspectCov(): handler(System.Exception+);
}
```

```
aspect DecisionCoverage {
    action AspectCov(): conditional(*);
}
```

We now use the *triangle problem* to demonstrate some of the scenarios in which our approach is used for specifying adequacy criteria as test coverage aspects. The triangle problem requires the program to read three numbers representing the lengths of three sides of a triangle and determine whether the triangle is equilateral, isosceles, or scalene.

Table 2: Coverage results

Package	Line of Code	Total number of join points	Coverage Percentage
NAnt.Core	11,356	3641	20.0
NAnt.DotNet	2253	784	15.9
NAnt.ZipTasks	230	65	100.0
NAnt.Console	155	41	53.4
SharpCVSLib	9245	2567	56.9

The interface of our *Triangle* class contains mutators and inspectors for the three edges of the triangle, a default constructor, a constructor that takes all three edges as arguments, and a function *Type* that returns the type of the triangle: equilateral, isosceles, scalene or illegal.

From now on, we present only the pointcut expression part of the coverage aspect. The adequacy criterion *Decision coverage only within the Triangle component*, will be expressed as the following pointcut expression in the coverage aspect:

```
conditional(*) && within(Triangle)
```

As shown in Table 1, the syntax of the within PCD is *within(TypePattern)*. It matches every join point from the code defined in a type in *TypePattern*. The richness of type patterns allows us to express quite interesting coverage concerns. For example, the pointcut *within(eos.*) && !within(eos.ast.*)* excludes the AST library from the coverage concern. Similarly, the pointcut *within(CodeObject+) && !within(CodeObject)* includes all the sub-types of CodeObject in the coverage but not the abstract class CodeObject itself.

After a modification to the method *Type*, the tester wants to perform decision coverage within the method; the tester can now specify this concern as the following pointcut expression:

```
conditional(*) && withincode(Triangle.Type)
```

To test the modifications in the function as well as the calls to the function, the pointcut expression can be modified as follows:

```
call(public * Triangle.Type(..))
```

```
|| (conditional(*) && withincode(Triangle.Type))
```

Another tester testing a crossword application that uses the triangle component wants to cover the coupling between the application and the component. This adequacy criterion can be expressed as the following pointcut in coverage aspect:

```
call(public * Triangle.*(..)) && !within(Triangle)
```

The criterion shown above means cover all calls to any method of the Triangle type which are not made within the type itself i.e. all calls from the application.

A problem with syntax-based pointcuts is that the original developer's intention to write them may not be entirely obvious later. Representing test adequacy criteria as pointcut expressions has similar problems. Even then, our approach is an advance over existing mechanism in the sense that the intentions, completely lost earlier, are now expressed as readable pointcut expressions.

Table 3: Space overhead of *AspectCov*

Package	Concern Coverage Criteria	Total number of join points instrumented	Executable size before instrumentation	Executable size after instrumentation
NAnt.Core	Handler (any)	78	385,024	417,792
NAnt.Core	iteration (any) conditional (any)	848	385,024	712,704
NAnt.DotNet	Handler (any)	8	90,112	94,208
NAnt.DotNet	iteration (any) conditional (any)	233	90,112	188,416

Table 4: Time overhead of *AspectCov*

Package	Concern Coverage Criteria	Total number of join points instrumented	Time to test before instrumentation (Secs)	Time to test after instrumentation (Secs)
NAnt.Core	Handler (any)	78	31.421	31.562
NAnt.Core	iteration (any) conditional (any)	848	31.421	40.281
NAnt.DotNet	Handler (any)	8	2.671	2.891
NAnt.DotNet	iteration (any) conditional (any)	233	2.671	3.206

To make the criteria more readable, these pointcut expressions may be annotated with comments describing the original intentions. Semantic pointcuts may provide a better solution, but they are beyond the state of the art and the scope of this paper.

5.2 Test Runs of *AspectCov*

To demonstrate the utility and to assess the space & time overhead of our approach, we have applied it to two significant, real open source C# projects, NAnt [45] and SharpCVSLib [10]. NAnt is a build tool similar to Ant but for the .NET framework, SharpCVSLib is a CVS client library for the C# language. The source code of these projects contains approximately 19,000 and 9,000 source lines, as of this writing, respectively. We selected these projects for evaluation based on their open source nature and the availability of test suites.

The NAnt project contains four sub-projects for which unit test-cases were available. We wanted to obtain decision and loop coverage assessments of these sub-projects. The criterion was represented as the following aspect:

```
aspect DecisionCoverage {
  action AspectCov():iteration(*)||conditional(*)
}
```

To measure coverage, we simply added the aspect to the code base of each sub-project. We provided the modified code to *AspectCov* to measure the coverage. Table 2 presents the results. The data show that the test sets provided with these open source projects

achieve coverage at best modest for most components, and plausibly not sufficient to meet a test of reasonableness.

We present detailed analysis of the test runs for two sub-projects, NAnt.Core and NAnt.DotNet, to assess the time and space overhead imposed by *AspectCov*. Two different coverage criterion were applied. The first criterion *iteration(*) || conditional(*)* measured the decision and loop coverage of the projects; the second criterion *handler(System.Exception+)* measured the error code coverage adequacy.

Table 3 presents the executable size before and after instrumentation. The executable size increases with the number of join points matching the coverage adequacy criterion. Currently, *AspectCov* weaves each matched join point. As mentioned before there are two optimizations possible. First, only the spanning set of the matched join points could be weaved. The spanning set is the minimal set of join points, execution of which implies execution of every matched join point. Second, complete weaving at these join points is not required to determine coverage adequacy. The coverage can be determined by just inserting a counter at the matched join points. Table 4 shows the time overhead of *AspectCov*. The time overhead also increases with the number of matched join points. The optimizations techniques described above will also reduce this overhead. In any case, the overheads were very reasonable in our experiments.

6. RELATED WORK

6.1 Program instrumentation

Instrumenting programs to obtain run time information is a well-studied area. Ball and Larus showed techniques for efficient path profiling and tracing programs [4], [5]. Their work describes algorithms to minimize the number of instrumentation points required in a program. Probert [48] describes a technique for optimal insertion of probes. There are other approaches for optimal program profiling as well, such as by Agrawal [1], Graham et al. [25], Knuth et al. [36], Ramamoorthy et al. [52], etc. A hardware-based approach for software profiling is Digital Continuous Profiling Infrastructure (DCPI) [14], [15], [63]. It uses performance counters on Alpha processors. All the approaches mentioned above emphasize increasing the efficiency of obtaining run time information when the set of program elements to be profiled are already specified. Our approach, on the other hand, provides means to select the program elements to be profiled. It appears hybrid approaches that combine the best of both worlds are possible, but exploration of this idea remains as future work. In our experience with Nant and SharpCVSLib, the performance impact was acceptable, in large part because of the highly selective instrumentation that our approach achieves.

6.2 Testing and code coverage analysis

There are many commercial and open source code coverage tools available, such as IBM Rational's Pure Coverage, Quilt [64], etc. These tools work well when the required code coverage criteria nicely fits in one of the options hard-coded inside the tools, however, they have no support for the crosscutting concern coverage that our approach provides.

The COverage MEasurement Tool (COMET) [26] developed at IBM Haifa Research Lab for system verification and micro-architecture verification has a flavor similar to that of our work.

The main idea is also to separate coverage criteria specification from the coverage analysis tool for enhanced flexibility. COMET uses a language containing first order temporal logic predicates along with simple arithmetic operators to define the model used for code coverage analysis. COMET is meant for functional coverage (functionality of the program) whereas our approach focuses on program-based selective coverage. COMET is used to test hardware designs ranging from systems to microprocessors and ASICs, whereas our approach is meant for software verification. The learning curve for COMET tends to be steep and writing models requires deep expertise [61].

Souter, Shepherd and Pollock [56] demonstrate the reduction in test suite execution overhead and increased precision in coverage information that will result if testing is performed with respect to concerns. They also present a framework for guiding selective instrumentation for scalable coverage analysis. Their framework allows the tester to build a concern using FEAT's graphical user interface [21]. The merit of our approach over their framework is that it avoids an explicit selection of program elements.

The approach presented by Tikir et al. [59] performs code coverage analysis by dynamic instrumentation of programs. Code coverage for Java (CC4J) [35] uses load time adaptation for code coverage analysis. As of now, our approach performs static instrumentation; however, the binding time for program instrumentation is orthogonal to our concerns.

Our goal is neither to minimize the number of instrumentation points required for code coverage analysis as in [4], [5] and [48] nor to minimize the overhead of analysis and overhead as in [59]. We intend to provide a mechanism in the form of a declarative, source code language for expressing test adequacy criteria. It does appear to be possible to achieve the benefits of selective and dynamic code coverage analysis together and to optimize the overhead of analysis, but testing this claim is out of scope.

6.3 AOP and Program Transformation

There are few approaches utilizing aspect-oriented programming for testing and verification purposes. Mahrenholz et al. [40] showed the use of AspectC++ for debugging and monitoring tasks. Ubayashi et al. [60], for example, described how to verify aspect-oriented programs using model checking and an AOP based model-checking framework. Their framework consists of multiple aspects, each specifying some property in the form of before and after advice. This approach however relies completely on execution of the weaved/instrumented program, and the properties that can be checked are limited to those that can be expressed in form of pre and post conditions.

There are other program transformation frameworks besides BCEL [8] and SOOT [57]. Kotik and Markosian [37] used REFINE for software analysis and test generation. One disadvantage of REFINE and a similar system, Gentle [62], is that both require learning a specialized transformation function notation that is quite different from the source or target language. Another similar program transformation system is TXL [11] that, unlike previous transformation systems, allows use of the target language. These meta-programming approaches to transformation are powerful but complex when compared to transformations using aspect-oriented approaches.

Open C++ [9] is similar to our approach in providing a framework to enable source code transformations, however, unlike Open C++

our approach is not limited to source code transformation, it recognizes the need for crosscutting activities in other software development processes. Even in the source code transformation domain of OpenC++, our approach facilitates crosscutting transformations that in turn enables modularized representation of crosscutting activities. The source code transformation enabled by Open C++ is more like insertions of tags in Java/.NET languages that serves as explicit request for source code transformation at a given point in the program.

6.4 Declare Constructs in AspectJ

AspectJ [3] provides three static crosscutting mechanisms with a flavor similar to the *actions* in our approach. The *declare parent* construct modifies the inheritance hierarchy of existing classes to declare a superclass or interface. The *declare error* and *declare warning* constructs identify certain usage patterns in the code base and emit compile time errors and warnings for policy enforcement. The policy enforcement enabled by these constructs is a primitive form of automated software inspection [17].

These constructs are similar to *actions* in the sense that they perform activities other than weaving, namely compile time editing of the code base and generation of messages. These constructs are, however, limited in the sense that the language designers of AspectJ need to add a new construct for each new software activity to be automated. Our *action* construct is far more general in that it allows the tool developers to choose their own *action identifier* and to define corresponding actions independent of the aspect language designer. In fact, the current AspectJ constructs become special cases of the *action* construct. In the future work section we describe how a more elaborate software inspection [19], [20] and smell detection [22] tool might be realizable using our language extensions and tool framework.

6.5 Attributes and Annotations

Attributes in .NET [44] languages, and annotations introduced in J2SE 5.0 [30], both called tags from now onwards, allow the designer to attach metadata to language constructs. A possible application of these tags is to examine the compiled assembly in case of .NET languages or .class/.jar file in case of J2SE for certain pre-defined metadata and perform *actions* based on that metadata. Burke [7] using JBoss AOP [31] and Shukla et al. [55] using C# attributes, among others, demonstrate aspect weaving as a possible action at points marked by specific metadata.

As pointed out by Kiczales [32], these tags are synergistic with aspect-oriented constructs. These tags explicitly request additional behavior at the join points where they are applied, where as pointcuts implicitly select join points. Currently our framework only allows join point selection using pointcuts, but in future extensions of the framework we hope to provide support for selection using tags.

We do not yet see as much value in representing test adequacy criteria using explicit tags. Doing so increases the burden on the tester to manually apply tags at points subject to test profiling.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented and demonstrated the potential utility of an approach to using aspect languages to express test adequacy criteria relative to crosscutting concerns. Our approach allows tester intentions to be represented abstractly within source

code. We provided one example of a white-box join point model, and a generalized *action* framework to support white-box testing tools. We evaluated potential utility by implementing language extensions, a framework, and a tool built on the framework. We assessed the expressiveness of the approach and the performance of the tool against two open source projects. One challenge now is to find a way to express a broader range of adequacy criteria. Pursuing the generalizations of aspect-oriented language model embodied in our approach and in Eos-T promises to help address this challenge.

We also made some progress toward extremes of aspect-oriented programming in two dimensions. The first is exposure of all semantically meaningful points in program execution as join points. The second is the ability to trigger arbitrary actions on code selected by pointcuts. The opening of compilation and execution to external behavioral modification has dangers, but also appears to create new possibilities for automating software development.

We anticipate two areas of future work: generalizing join point models, and identifying other development activities amenable that can potentially benefit from our approach. The join point model discussed in the current work provides early evidence that white-box join point models might have real value in practice. To express more complex coverage criteria, such as multiple condition/decision coverage, even richer models are needed.

In the future, we will also consider software *concern inspection* as an activity to automate. Software inspection, introduced by Fagan [19], [20] and reported by others [23], [53] is a well-known technique for improving software quality. It involves examining the software artifacts for aspects known to be potentially problematic. Manual formal code reviews and structured walk-through are costly processes. The costs sometimes result in inspections that are not performed well or sometimes even not at all. Tools for automated code inspection try to address this problem by relieving the programmers of the manual inspection burden. One such class of automatic code inspection tools [17] used to detect code *smells* [22] automatically finds and reports bad design and programming styles.

Similar to coverage analysis tools, current code inspection tools limit the flexibility to the set of predefined bad design and programming styles hard-coded in the tool. The developer cannot specify smells. This verification process is also a candidate for the use of a declarative language to specify smell patterns in the program that can be detected later using tools that could be implemented using our framework. A typical smell in OO program is the use of switch statements in a method. This smell can be identified using simple pointcut expression:

```
conditional(switch) && withincode(* *.*(..))
```

Similar smells can be represented as pointcut expressions and associated with a *CodeInspection* action. This flexibility enables organization, team, and individual level customization of the automatic inspection process. A tool similar to our code coverage analyzer can be implemented for this task using our framework.

8. ACKNOWLEDGMENTS

We would like to thank Wolfram Schulte for valuable discussion about the coverage analysis practices inside Microsoft and Michael Tashbook for providing valuable comments on the draft.

This work was supported in part by the National Science Foundation under grant ITR-0086003.

9. REFERENCES

- [1] Agrawal, H., "Dominators, Super Blocks and Program Coverage", *POPL* 94, Portland, Oregon, pp. 25 – 34.
- [2] AspectC++ Homepage: <http://www.aspectc.org>.
- [3] AspectJ Homepage: <http://www.eclipse.org/aspectj>.
- [4] Ball, T., and Larus, J., "Optimally Profiling and Tracing Programs", *ACM Transactions on Programming Languages and Systems*, Vol 16, no. 4, July 1994, pp. 1319 – 1360.
- [5] Ball, T., and Larus, J., "Efficient Path Profiling", 29th Annual *IEEE/ACM International Symposium on Microarchitecture*, Paris, France, pp. 46 – 57.
- [6] Boehm, B., "Software and its impact: a quantitative assessment", *Dataation* 19(May 1973), pp. 48 – 59.
- [7] Burke B., "Aspect-Oriented Annotations", <http://onjava.com> article dated Aug 25, 2004.
- [8] Byte Code Engineering Library (BCEL): <http://jakarta.apache.org/bcel/>
- [9] Chiba, S., "A Metaobject Protocol for C++", In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), page 285-299, October 1995.
- [10] CVS client library for C#: <http://sharpcvslib.sourceforge.net/>
- [11] Cordy, J. R., Halpern-Hamu, C. D., and Promislow, E., "TXL: a rapid prototyping system for programming language dialects", *Computer Languages* 16(1) (1991): pp. 97 – 107.
- [12] Cordy, J.R., and Shukla, M., "Practical Metaprogramming", Proc. *CASCON '92*, IBM Centre for Advanced Studies 1992 Conference, Toronto, November 1992, pp. 215 – 224.
- [13] Dahl, O. J. , Dijkstra, E. W., and Hoare, C. A. R. "Structured Programming", Academic Press, New York, 1972.
- [14] Dean, J., Hicks, J.E., Waldspurger, C.A., Weihl, W. E., and Chrysos, G., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors", Proceedings of the Thirtieth Annual *IEEE/ACM International Symposium on Microarchitecture*, December 1-3, 1997, Research Triangle Park, North Carolina, USA, pp. 292 – 302.
- [15] Dean, J., Waldspurger, C. A., and Weihl, W.E., "Transparent, Low-Overhead Profiling on Modern Processors," Workshop on *Profile and Feedback-Directed Compilation* held in conjunction with PACT' 98. Oct. 1998, Paris, France.
- [16] Elbaum, S., Gable, D., Rothermel, G., "The impact of software evolution on code coverage information", *IEEE International conference on Software Maintenance*, 2001, pp. 170 – 179.
- [17] Emden, E. V., and Moonen, L., "Java Quality Assurance by Detecting Code Smells", Proceedings of the *9th Working Conference on Reverse Engineering*, IEEE Computer Society Press, Oct 2002, pp. 97 – 108.
- [18] Eos Homepage: <http://www.cs.virginia.edu/~eos>
- [19] Fagan, M.E., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, 15, 3, 1976, pp. 182 – 211.
- [20] Fagan, M.E., "Advances in Software Inspections", *IEEE Transactions in Software Engineering* SE-12, 7 (July) 1986, pp. 744 – 751.
- [21] FEAT: <http://www.cs.ubc.ca/labs/spl/projects/feat/>
- [22] Fowler, M., "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999.
- [23] Gilb, T., and Graham, D., "Software Inspection", Addison-Wesley, 1993.
- [24] Goodenough, J. B., Gerhart, S. L., "Toward a Theory of Test Data selection", *IEEE Transactions on Software Engineering*, 1(2), pp. 156 – 173, 1975.
- [25] Graham, S.L., Kessler, P.B., and McKusick, M.K., "An execution profiler for modular programs", *Software Practice and Experience*, 13: pp. 671 – 685, 1983.
- [26] Grinwald, R., Harel, E., Orgad, M., Ur S., and Ziv, A., "User Defined Coverage – A Tool Supported Methodology for Design Verification", *Design Automation Conference*, 1998, pp. 158 – 163.
- [27] Harrold, M. J., "Testing: A Roadmap", In Future of Software Engineering, 22nd *International Conference on Software Engineering*, June 2000, pp. 61 – 72.
- [28] Harrold M. J., and Rothermel, G., "Aristotle: A System for Research On and Development of Program-Analysis-Based Tools", Technical Report OSU-CISRC-3/97-TR17, *Department of Computer and Information Science, The Ohio State University*, March 1997.
- [29] Hong, Z., Patrick A. V. H., and John H. R. M., "Software unit test coverage and adequacy", *ACM Computing Surveys*, 29(4): pp. 366 – 427, 1997.
- [30] Java: <http://java.sun.com>
- [31] JBoss AOP: <http://www.jboss.org/products/aop>
- [32] Kiczales, G., "The More the Merrier", *Software Development Magazine*, Sept 2004.
- [33] Kiczales, G., "The fun has just begun", Key note address of *International Conference on Aspect-Oriented Software Development*, Boston, MA, 2003.
- [34] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., "Aspect-oriented programming," in Proceedings of the *European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlang, Lecture Notes on Computer Science 1241, June 1997, pp. 220 – 242.
- [35] Kniessel, G., Austermann, M., "CC4J - Code Coverage for Java - A Load-Time Adaptation Success Story", In *Component Deployment - IFIP/ACM Working Conference*, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings, Springer LNCS 2370, pp. 155 – 169, 2002.
- [36] Knuth, D., E., and Stevenson, F., R., "Optimal measurement points for program frequency counts", *BIT*, 13: pp. 313 – 322, 1973.

- [37] Kotik, G., Markosian L., "Automating software analysis and testing using a program transformation system", Proceedings of the *ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, Key West, Florida, United States, pp. 75 – 84, 1989.
- [38] Lamping, J., "The role of the base in aspect-oriented programming", First Workshop on *Multi-Dimensional Separation of Concerns in Object-oriented Systems* (at OOPSLA '99).
- [39] Larus, J., "Whole Program Paths," *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99)*, May 1999, Atlanta Georgia.
- [40] Mahrenholz, D., Spinczyk, O., Schröder-Preikschat, W., "Program Instrumentation for Debugging and Monitoring with AspectC++", The 5th *IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Washington DC, USA, April 29 - May 1, 2002, pp. 249 – 256.
- [41] Marrè, M., and Bertolino, A., "Using Spanning Sets for Coverage Testing", *IEEE Transactions on Software Engineering*, Vol. 29, No. 11, November 2003, pp. 974 – 984.
- [42] Masuhara, H., and Kiczales G., "Modular Crosscutting in Aspect-Oriented Mechanisms", *ECOOP 2003*, Darmstadt, Germany, July 2003.
- [43] Microsoft. C# Specification Homepage. <http://msdn.microsoft.com/net/ecma>
- [44] Microsoft .Net: <http://www.microsoft.com/net>
- [45] NAnt homepage: <http://nant.sourceforge.net/>
- [46] NUnit homepage: <http://nunit.sourceforge.net/>
- [47] Ossher, H., and Tarr, P., "Operation-Level Composition: A Case in (Join) Point", Workshop on aspect-oriented programming, ECOOP 1998.
- [48] Probert, R. L., "Optimal Insertion of Software Probes in Well-Delimited Programs," *IEEE Transactions on Software Engineering*, January, 1981, pp. 34-42.
- [49] Rational PureCoverage: <http://www.ibm.com>
- [50] Rajan, H., and Sullivan, K., "Eos: Instance-Level Aspects for Integrated System Design", *2003 Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 03)*, (Helsinki, Finland, Sept 2003).
- [51] Rajan, H., and Sullivan, K., "Classpects: Unifying Aspect- and Object-Oriented Language Design", Technical Report CS-2004-21, Department of Computer Science, University of Virginia, Sept 2004.
- [52] Ramamoorthy, C. V., Kim, K.H., and Chen. W. T., "Optimal placement of software monitors aiding systematic testing." *IEEE Transactions on Software Engineering*, SE-1(4):403-411, Dec. 1975.
- [53] Russell, G. W., "Experience with inspection in ultralarge-scale developments." *IEEE Software*, 8(1):25-31, 1991.
- [54] Schulte, W., Personal communication with Hridesh Rajan at Microsoft Research
- [55] Shukla, D., Fell, S., and Sells, C., "Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse", *MSDN Magazine*, March 2002.
- [56] Souter, A. L., Shepherd, D., and Pollock, L L., "Concern-based Testing," *IEEE International Conference on Software Maintenance*, September, 2003
- [57] Soot: a Java Optimization Framework: <http://www.sable.mcgill.ca/soot/>
- [58] Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr. S.M., "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999.
- [59] Tikir MM, Hollingsworth JK., "Efficient instrumentation for code coverage testing.", *ACM. Sigsoft Software Engineering Notes*, vol.27, no.4, July 2002, pp.86-96. USA.
- [60] Ubayashi, N., Tamai, T., "Aspect-oriented programming with model checking", Proceedings of the 1st international conference on Aspect-oriented software development, Enschede, The Netherlands, 148 - 154, 2002.
- [61] Ur, S., Ziv, A., "Of-The-Shelf Vs. Custom Made Coverage Models, Which is the one for You? ", *STAR 98* May 1998.
- [62] Vollmer, J., "Experiences with Gentle: efficient compiler construction based on logic programming." *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP 91)*, Springer Verlag Lecture Notes in Computer Science 528(1991): 425-426.
- [63] Weihl, W.E., "CPI: Continous Profiling Infrastructure", *DIGITAL Forefront Magazine*, Winter 1997, pages 27-28.
- [64] Quilt: <http://quilt.sourceforge.net/>