Generalizing AOP for Aspect-Oriented Testing

Hridesh 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 profiling program execution to assess the adequacy of a test set, a challenge is to select the code to be included in the coverage assessment. Current mechanisms for doing this are coarse-grained, assume traditional concepts of modularity, require tedious and error-prone manual selection, and leave the tester's intent implicit in the input provided to the testing tools. The aspect-oriented constructs of languages such as AspectJ promise to help ease and extend our ability to select the code to be included in a test adequacy criterion and to document the tester's intent within the source code itself. Our contribution is a language-centric approach to automated test adequacy analysis that we call *concern coverage.* We claim that our approach enables explicit, precise, abstract, and machine-readable representation of the tester's intent and that it can ease testing by eliminating the need for manual selection and explicit maintenance of test adequacy criteria.

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
The problem that we address in this paper is that the state of the art in selecting the code that is to be included in an adequacy criterion, and to be monitored at runtime to measure adequacy, falls short in three ways. First, current mechanisms are too coarse-grained and too closely based on traditional modules to support fine-grained selection of scattered code. Second, these mechanisms require tedious, error-prone manual selection of included or excluded elements, usually through the GUI of a testing tool. Commercial tools for code coverage analysis such as IBM Rational's Pure Coverage [48], and open source tools such as Quilt [63], for example, provide control over exclusion or inclusion of code at the file, class, and function level only, and only by manual enumeration through a graphical user interface. Third, expressing adequacy criteria by selecting modules alone does not clearly express the intent of the tester, potentially complicating downstream system maintenance and evolution.

To address these shortcomings we present an approach and a system for what we call *concern coverage.* The idea is to express the test adequacy criterion in an abstract, declarative form as a part of the code base to be tested. The criterion is represented using aspect-oriented [4][34] pointcut constructs in the style of AspectJ [3]. The tool then automates the rest of the analysis.

Our language-centric approach addresses the three problems enumerated above. First, the underlying declarative language allows expression of complex, crosscutting adequacy criteria. Second, the tool takes the declarative representation as input and automates the analysis, relieving the tester of the costly and error-prone task of manual code selection. Third, the representation of adequacy criterion clearly expresses the tester's intent as part of the source code itself.

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

2. BACKGROUND

2.1 Aspect-oriented programming
To make this work self-contained, we briefly review basic concepts in the dominant aspect-oriented model. The central idea is that aspects enable the modular representation of crosscutting concerns. A concern is a dimension in which a design decision is made, and is crosscutting if it cannot be realized in traditional object-oriented designs except with scattered and tangled code [38]. By scattered we mean not localized in a module but fragmented across a system. By tangled we mean intermingled with code for other concerns [33].

Conference'05, Month 1–2, 2005, City, State, Country.

Dominant aspect languages add five key constructs to the object-oriented model: join points, pointcuts, advice, introductions (not discussed in any detail in this paper), and aspects. We provide a simple example to make the points concrete.

```
1 aspect Tracing {
2   pointcut tracedCall():
3       call(* *(..));
4   before(): tracedCall() {
5       /* Trace the call */
6   }
7 }
```

An aspect (lines 1-7), performs behavioral modification to program execution event exposed by the language definition. These events are called join points. The execution of a method in the program in which the *Tracing* aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification—here, any call to any method. An advice (see lines 4-6) serves as a *before, after,* or *around* method to effect such a modification at each join point selected by a pointcut [50].

2.2 Code Coverage Analysis

Exhaustive testing [7] refers to exercising all possible inputs of a program in order to prove the absence of faults. Unfortunately, in the typical case of vast or infinite input spaces, exhaustive testing is infeasible [24], and any sub-set of an exhaustive test set can only show the presence of bugs not their absence [13]. Even though exhaustive testing is not always possible, thorough testing can provide a basis for sufficient confidence in program behavior. A *test adequacy criterion* is a predicate defining what properties of a program must be exercised to constitute a thorough test [24]. Adequacy criteria help determine when testing may cease [29].

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.*

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.*

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 [53] observed the need to represent such selective and crosscutting adequacy criteria in large projects inside Microsoft, where the tester is only interested in testing selected parts of a large code base. At Microsoft the need is met using relatively ad hoc techniques.

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. This approach does not offer enough flexibility. 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 suffer from what has been called the tyranny of dominant decomposition [57]. It is not surprising that the existing mechanisms to measure test adequacy exhibit similar problems as non-modular crosscutting concerns in code.

Assessing the satisfaction of test adequacy criteria requires monitoring of key points in program execution. As mentioned, in aspect-oriented terminology, a join point is a point in the execution of a program. An adequacy criterion, like a pointcut, selects a subset of join points. The idea behind our approach is to express the code to include in an adequacy criterion using declarative, aspect-oriented pointcut constructs. For example, one could express the test adequacy 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 the following aspect, and then expect the tool support to automate the rest of the code 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 the aspect body.

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

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

Second, these aspects relieve the tester of the tedious and error-prone task of manually selecting and reselecting files, packages, and functions to be included. 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 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 two ways. First, they lack join point models at a granularity fine enough to support white-box testing. The problem is not merely accidental. Rather, it reflects a fundamental decision made by the language designers, that the only join points that should be exposed are at module boundaries (e.g., calls to methods). Such a *black box join point model* is inadequate to support *white box* test set assessment. For example, no aspect languages that we know of, except for experimental versions of our own Eos language, expose branches as join points. This shortcoming would preclude precise, abstract expression of crosscutting branch coverage criteria. We address this problem by introducing the idea of generalized (white-box) join point models and pointcut languages. As an example, in this paper, we present an aspect language design that exposes branches as join points.

Second, with the exception of a few ad hoc mechanisms, most aspect languages support only one action at join points selected by aspects, namely weaving of advice to modify normal 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 proposed by Ball and Larus [5][6], are likely to be superior in many cases. We address this problem by introducing the idea of generalized advice: directives that can be processed by the supporting tools to cause actions other than mere weaving of advice code at designated join points. An example would be the application of the Ball-Larus algorithm to instrument the join points for profiling or tracing.

To evaluate the feasibility of these ideas, we have designed and implemented Eos-T (where *T* is for testing), a version of Eos [18], [49] with added support for generalized join point and pointcut models and generalized advice. Eos is an aspect-oriented version of C# [43] a .NET [44] language. It 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 by a fully functional compiler, comprising approximately 40,000 source 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 rest of this section, we discuss the need for the language model extension to support our ideas and describe how the extensions incorporated in Eos-T 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 seemed like an effective vehicle to select program elements, execution of which contributes to the satisfaction of an adequacy criterion. For example, the *FunCov* criterion discussed earlier could be represented by the simple pointcut highlighted in the code below.

```
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 model and pointcut sub-languages of the current aspect languages are not fine grained enough to express test adequacy criteria at a granularity needed for testing. A hypothetical language L, which supports only method execution as join points will limit the possible adequacy criteria to some variant of function coverage. Extending the join point model of L to include the exception handlers will allow us to express the error code coverage as well. Further, to express the statement coverage criteria we will need the join point model of L to contain

individual statements. The AspectJ language model, clearly the most mature and well-developed model, recognizes method call & execution, field get and set, exceptions, control flow, etc., as join points but it does not recognize basic blocks or branches as join points precluding the possibility to do decision coverage.

Second, most aspect languages only support weaving of advice code at a join point selected by an aspect, but not other useful actions. Apart from weaving advice, one might want to schedule these points for code coverage analysis, schedule these 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 thus envision decoupling the notion of pointcut expressions from the actions to be taken at the designated join points, with advising as a special case.

In the next subsection, we will discuss the language extensions to Eos to enable representation of test adequacy criteria as aspects.

3.2 Eos-T and its Extended Join Point Model

To provide a basis for an early evaluation of the potential value of a generalized join point model in supporting aspect-oriented test coverage analysis, Eos-T extends the join point model of Eos and AspectJ 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 beginning and the end of a basic block in a program. Each of these join points is discussed in detail in the rest of this sub-section.

A conditional statement in C# is an *if* statement or a *switch* statement, whereas an iteration statement is either of the *while*, *do-while*, *for*, and *foreach* statements. The Eos-T pointcut sublanguage allows selecting these join points using the pointcut designators (PCDs) *conditional* and *iteration*. 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
```

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

PCD	Explanation
call(Signature)	Matches every methods call site matched by the signature
Execution(Signature)	Matches execution of every method matched by the signature
fget(Signature)	Matches variable read for variables matching the signature
fset(Signature)	Matches variable write for variables matching the signature
Initialization(Signature)	Matches object initialization/constructor calls matching the signature
Handler(TypePattern)	Matches exception handlers catching an exception of type matched by the pattern
Adviceexecution()	Matches execution of every advice
Within(TypePattern)	Matches each join point within the type matched by the pattern
withincode(Signature)	Matches every join point within the method matched by the signature
Pget(Signature)	Matches property read for variables matching the signature
Pset(Signature)	Matches property write for variables matching the signature
Conditional(if/switch)	Matches conditional join points (if and switch statements)
Iteration(while/dowhile/for /foreach)	Matches iteration join points (while, do-while, for and for each statements)

In addition to these new PCDs, EOS-T also 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 shown in the table are anchored to an interface element and they select a group of join points with respect to that interface element. For example, the call PCD is anchored to an interface element, method name, and it picks out all the calls to the specified method regardless of where the call is made from. As can be observed, the *conditional* and the *iteration* PCDs are not anchored to any named interface element. They are instead anchored to the programming language constructs and select all the program elements that belong to that 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 expressive 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.

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 a range of new possibilities for expressing actions across the software lifecycle as part of the source code itself: in a precise, abstract, *modular* and *composable* form. The list of possible action includes but is not limited to code coverage analysis, code inspection, bug monitoring, automated document generation, etc.

This approach promises two potentially significant benefits. First, representing actions explicitly and abstractly within the code base promises to increase the reliability of certain lifecycle activities, by representing them in the place most visited by real developers, in a form in which they are easily subject to automation. Second, as opposed to other representations of the same information, this representation does not require separate maintenance efforts.

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. 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 will dramatically reduce 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. Tools recognizing these extensions can automatically perform these embedded actions. In the next Section, we describe our framework for implementing tools that recognize these extensions.

4. FRAMEWORK

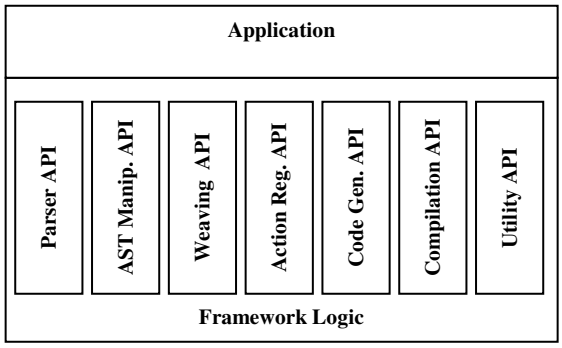

Fig. 1 Framework overview

As illustrated in Figure 1, our framework provides application programming interfaces (APIs) for source code parsing[1], 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.

As opposed to other frameworks for implementing verification tools, e.g., Aristotle [28], our framework recognizes crosscutting concerns and offers an interface to perform crosscutting actions. Java byte code transformation tools such as BCEL [9], SOOT [56], 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 a coverage tool supporting *concern coverage* as a test adequacy criterion.

5. THE *ASPECTCOV* TESTING TOOL

Our tool for code 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.

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

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. For example, a 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 simulation and speeding the verification 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. On construction, it registers itself with the framework to receive notifications for the action *AspectCov*.
2. In the initialization stage, it collects all the *AspectCov* actions. As we have 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. In the run stage, our tool selects a union[2] of these sub-sets of join points that need to be instrumented.
4. It generates an aspect containing handcrafted advice to serialize the reflective information at the join point and send it to the code coverage information collection server running at a well-known port. To optimize it further, 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 invoke the handcrafted advice in the aspect trace. Any of the optimal instrumentation algorithms can be plugged in to actually instrument the code.
6. The abstract syntax tree of the instrumented program is compiled into assembly[3].
7. This instrumented assembly is loaded into the NUnit unit-testing framework.
8. 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, change across components, and change across tests described by Elbaum, Gable, and Rothermel [16] to give an idea of the change in coverage between versions.

[2] More sophisticated mechanisms such as [1], [5], [6], [25], [36], [47], [51], etc. can also be used.

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

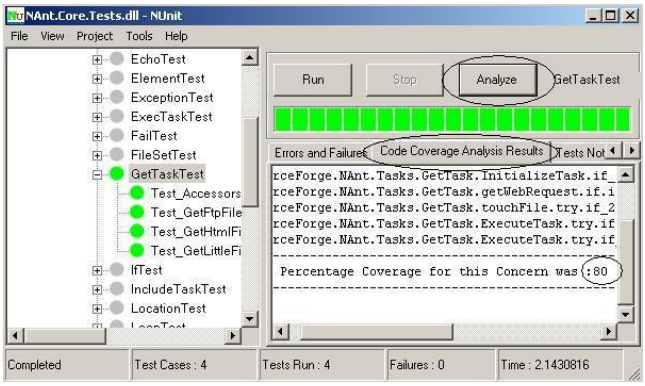

Fig. 2 Screen shot of *AspectCov* integrated with NUnit.

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 are using the NUnit framework [46] as a testing tool; however, our approach can be easily extended to other testing infrastructures. Figure 2 shows the bottom line coverage 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. This overhead is minimal given the simplicity of the aspect.
2. Compilation overhead: The code base and the coverage aspect need to be compiled. Currently our approach works at the source code level so recompiling the source 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, the auto-generated coverage advice is invoked. This coverage advice can be inlined for further optimization. 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 [5] 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. Overall, decisions in this dimension are orthogonal to the central contribution of this work in the method and system for concern coverage based on two generalizations of the aspect-oriented programming model.

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 also co-exist with aspects.

Consider the adequacy criterion *Every modification to the value field of the class Model should be covered*. 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 the task.

```
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* (one of the most common examples in the testing literature) 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.

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.

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

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 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.

5.2 Test Runs of *AspectCov*

To assess the practicality 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.

Table 3: Space overhead of *AspectCov*

Package	Concern Coverage Criteria	Total number of join points instrum-ented	Executable size before instrum-entation	Executable size after instrum-entation
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 instrum-ented	Time to test before instrum-entation (Secs)	Time to test after instrum-entation (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

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

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

To measure the 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 branch 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 appear very reasonable.

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 [5], [6]. Their work describes algorithms to minimize the number of instrumentation points required in a program. Probert [47] 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. [51], etc. A hardware-based approach for software profiling is Digital Continuous Profiling Infrastructure (DCPI) [14], [15], and [62] that uses performance counters on the 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 [63], 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 needs expertise [60].

Souter, Shepherd and Pollock [55] 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. [58] performs code coverage analysis by dynamic instrumentation of the program. 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 [5], [6] and [47] nor to minimize the overhead of analysis and overhead as in [58]. We intend to provide a mechanism in the form of a declarative, source code language for expressing test adequacy criteria. It is, however, possible to achieve the benefits of selective and dynamic code coverage analysis together and optimize the overhead of analysis, but it is beyond the scope of this work.

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. [59], 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 [9] and SOOT [56]. Kotik and Markosian [37] used REFINE for software analysis and test generation. One disadvantage of REFINE and a similar system, Gentle [61], 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.

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 activity 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 [8] using JBoss AOP [31] and Shukla et al. [54] 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 see as much value in representing test adequacy criteria using explicit tags. Doing so increases the burden on the tester to manually apply tag at points subject to test profiling.

7. CONCLUSION

In this paper, we have presented and demonstrated significant potential utility in 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 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 (e.g., inspections, document generation, etc) on code selected by pointcuts. The opening of compilation and execution to external behavioral modification clearly has dangers, but also appears to create new possibilities for automating software development.

8. FUTURE WORK

Software inspection, introduced by Fagan [19], [20] and reported by others [23], [52] is a well-known technique for improving software quality. It involves examining the software artifacts for aspects known to be potentially problematic. Manual software inspection techniques such as formal code reviews and structured walk-through are formal, labor-intensive processes guided by

well-defined rules. 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, automatic 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 her own specific 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.

[Note to reviewers: The Eos-T compiler is freely available from www.cs.virginia.edu/~eos for research and teaching.]

9. 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.

10. 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] Aspect Oriented Software Development: http://www.aosd.net.

[5] Ball, T., and Larus, J., “Optimally Profilling and Tracing Programs”, ACM Transactions on Programming Languages and Systems, Vol 16, no. 4, July 1994, pp 1319 – 1360.

[6] Ball, T., and Larus, J.,"Efficient Path Profiling," *29th Annual* IEEE/ACM International Symposium on Microarchitecture, Paris, France, pp. 46-57.

[7] Boehm, B., “Software and its impact: a quantitative assessment”, Datamation 19(May 1973), 48-59.

[8] Burke B., “Aspect-Oriented Annotations”, http://onjava.com article dated Aug 25, 2004.

[9] Byte Code Engineering Library (BCEL): http://jakarta.apache.org/bcel/

[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): 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.

[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. October 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.

[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.

[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), 156-173, 1975.

[25] Graham, S.L., Kessler, P.B., and McKusick, M.K.,. An execution profiler for modular programs. Software Practice and Experience, 13:671-685, 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.

[28] Harrold M. J., and Rothermel, G., "Aristotle: A System for Research On and Developement 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):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.

[35] Kniesel, G., Austermann, M., "CC4J - Code Coverage for Java - A Load-Time Adaptation Success Story", In Component Deployment - IFIP/ACM Working Confrerence, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings, Springer LNCS 2370, pp. 155-169, 2002. ISBN 3-540-43847-5.

[36] Knuth, D., E., and Stevenson, F., R., Optimal measurement points for program frequency counts. BIT, 13:313-322, 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, Pages: 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.

[41] Marrè, M., and Bertolino, A.,"Using Spanning Sets for Coverage Testing", IEEE Transactions on Software Engineering, Vol. 29, No. 11, November 2003, p. 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] Probert, R. L., "Optimal Insertion of Software Probes in Well-Delimited Programs," *IEEE Transactions on Software Engineering*, January, 1981, pp. 34-42.

[48] Rational PureCoverage: http://www.ibm.com

[49] 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).

[50] 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.

[51] 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.

[52] Russell, G. W., "Experience with inspection in ultralarge-scale developments.", IEEE Software, 8(1):25-31, 1991.

[53] Schulte, W., Personal communication with Hridesh Rajan at Microsoft Research

[54] Shukla, D., Fell, S., and Sells, C., “Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse”, MSDN Magazine, March 2002.

[55] Souter, A. L., Shepherd, D., and Pollock, L L., "Concern-based Testing," *IEEE International Conference on Software Maintenance*, September, 2003

[56] Soot: a Java Optimization Framework: http://www.sable.mcgill.ca/soot/

[57] 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.

[58] 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.

[59] 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.

[60] Ur, S., Ziv, A., “Of-The-Shelf Vs. Custom Made Coverage Models, Which is the one for You? ”, STAR 98 May 1998.

[61] 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.

[62] Weihl, W.E., "CPI: Continous Profiling Infrastructure", DIGITAL Forefront Magazine, Winter 1997, pages 27-28.

[63] Quilt: http://quilt.sourceforge.net/