The Computer Journal Advance Access published May 25, 2007

doi:10.1093/comjnl/bxm021

A Survey of Coverage-Based Testing Tools

Qian Yang*, J. Jenny Li and David M. Weiss

Avaya Labs Research, 233 Mt Airy Road, Basking Ridge, NJ 07920, USA
**Corresponding author: yangqian@research.avayalabs.com*

Test coverage is sometimes used to measure how thoroughly software is tested and developers and vendors sometimes use it to indicate their confidence in the readiness of their software. This survey studies and compares 17 coverage-based testing tools primarily focusing on, but not restricted to, coverage measurement. We also survey features such as program prioritization for testing, assistance in debugging, automatic generation of test cases and customization of test reports. Such features make tools more useful and practical, especially for large-scale, commercial software applications. Our initial motivations were both to understand the available test coverage tools and to compare them to a tool that we have developed, called eXVantage (a tool suite that includes code coverage testing, debugging, performance profiling and reporting). Our study shows that each tool has some unique features tailored to its application domains. The readers may use this study to help pick the right coverage testing tools for their needs and environment. This paper is also valuable to those who are new to the practice and the art of software coverage testing, as well as those who want to understand the gap between industry and academia.

Keywords: code coverage, coverage-based testing tool, prioritization, test case generation, dominator analysis

Received 30 August 2006; revised 11 April 2007

1. INTRODUCTION

In strongly competitive industries where software is the only or a key component of a product, customer satisfaction may be highly correlated with software quality. In some industries, such as automobile manufacturing or avionics, software defects may jeopardize human life, introduce considerable negative financial consequences and ruin customer relations. As a result, there is a continuing pressure in such industries to improve software quality, particularly to have quantitative evidence of such improvement.

Software testing is a practice often used to indicate software quality. Because it usually does not contribute directly to adding new features, testing may be considered overhead that must be as effective as possible. Indeed, testing is a very labor and resource intensive process that often accounts for between 40 and 80% of the total cost of software development [1]. Understanding the time and resources that should be allocated to testing involves a trade-off among budget, time and quality [2]. In unsystematic development processes, testing continues until time runs out [3]. Where development is more systematic, organizations seek measures of testing completeness and goodness to establish test completion criteria. Code coverage is one such measure [4].

Our objective in this paper is to help practitioners' select coverage tools appropriate for their needs and environment, and also to give some indication of the state of the technology in coverage tools. This paper is also valuable to those who are new to the practice and the art of software coverage testing, as well as those who want to understand the gap between industry and academia.

We first noticed the increased interest in code coverage tools several years ago in our software development organizations. In project assessments [5], internal symposia and informal discussions, we discovered several significant issues as follows.

(1) Developers and testers had no good way of knowing how much of their code had been covered in testing. Development managers were particularly interested in knowing how well the tested code was when it passed through development milestones such as completion of unit testing, integration testing or system testing.
(2) The cycle of change, build and test was generally inefficient and ineffective. There were too many manual

steps in the cycle and too many defects slipped through to the field. (Exact numbers are proprietary, but dissatisfaction was driven by schedule slippage and the incidence of field errors evidenced in customer complaints.)

(3) Developers were so enmeshed in firefighting, that they did not have time to search for tools to automate their processes [6].

Accordingly, we set out to help introduce better build and test processes using automation. As one element of our approach, we focused on code coverage, a measure that was easy to understand by developers and managers, and that had some intuitive appeal, even if the underlying theory of its effectiveness in detecting defects was lacking. Note that we helped to introduce other techniques as well, such as architecture reviews [7], but we have limited the scope of this paper to code coverage tools and the features they offer, since such tools were a major draw for our working software developers and their managers. This focus also drove our research into how such tools might be improved, leading to extensions and improvement of eXVantage,[1] our in-house coverage tool [8].

This paradigm of observing and working closely with software engineers as a way to drive our research is in the tradition of Pasteur's Quadrant [9]. Often, one obtains surprising insights using such a paradigm. For example, we initially thought that developers were not interested in off-line overhead, such as the time and space needed to instrument their code to obtain coverage results. This turned out to be untrue in several of our development organizations, which are engaged in tight optimization of their development resources, such as processor time and disk space.

The results and observations presented in this paper are driven by our empirical observations of our developers and their needs, and by discussions, driven by those same observations, with developers of commercial code coverage tools. For example, presenting our development community with a survey and comparison of code coverage tools and their applicability to real-time systems helped them to decide which tools would be best for them. We also limited our analysis of tools only to those available to our development community, i.e. commercially available tools, open source tools and our in-house tool, eXVantage.

Based on this empirically derived methodology, we focus this survey on tools that measure testing coverage. Finding an effective and efficient software-testing tool could be a major benefit for a project or a company. Yet, there is no single test tool suitable for all possible systems and industry sectors. Deciding what criteria to apply when selecting a specific tool for a project is tricky. For example, some tools integrate seamlessly with your choice of IDE (e.g. Eclipse) and provide user-friendly interfaces to ease unit testing in the development stage, but have scalability issues. Those tools are suitable for a small project, but not a large-scale commercial application that sometimes includes a large percentage of legacy code. Other tools provide fine testing granularity, but the performance overhead inevitably prevents them from being useful in real-time or embedded systems.

We selected only test tools with code coverage capabilities. We found 17 tools, including one in-house, that fit our category and for which information is available in the public domain. Besides studying the 16 external coverage-based test tools, our goal was also to evaluate our in-house tool suite, eXVantage (a tool suite for code coverage testing, debugging and performance profiling). We contacted all companies whose tools are included in our comparison, and had further direct conversations with several of them. For example, one question we asked is how much instrumentation overhead tools incurred. We also sent this paper to all vendors for review. The Appendix contains the request we sent to all the companies, including the questions we asked.

In addition to coverage measurement, test tools often provide other functions, such as rule checking, profiling and debugging assistance. We exclude tools that only perform static analysis, load testing or functional testing without coverage measurement. We compiled descriptions of each test tool based on the information in the public domain. Our descriptions were reviewed by some but not all of the venders providing the software, at their discretion. The descriptions are factored into several different categories covering important functions and features of the testing tools.

The rest of this paper is organized as follows. Section 2 provides an overview of coverage with its pros and cons. Section 3 discusses some important aspects of coverage measurement, including programming languages, overhead and auxiliary features to code coverage. Section 4 presents various coverage criteria supported by each tool. Section 5 discusses different approaches to prioritization, illustrating how some of the tools use prioritization to get 'good test cases' in an efficient way for complex software suites. Section 6 covers automating test case generation in the context of coverage-based testing, and covers the user interfaces of the tools. GUI and batch-mode versions provide different benefits. A comprehensive reporting component facilitates communication among the team members or to the customers. Our summary appears in Section 7 along with a table summarizing the major aspects of our analysis.

2. OVERVIEW OF COVERAGE

Coverage-based testing measures the percentage of the software that is exercised in the process of testing. It is applicable to any stage of testing, including unit testing, integration testing or system testing. Test coverage can be based on a

[1]eXVantage stands for eXtreme Visual-Aid Novel Testing And GEneration. eXVantage is a testing tool jointly developed by Avaya Labs Research and the University of Texas at Dallas.

functional specification (black-box testing) or on internal program structure (white-box testing). Because functional specification-based coverage relies on the availability of specifications, structure-based coverage is more commonly used. Such testing can measure coverage at various granularities, including statements, lines, blocks, conditions, methods and classes. It provides a way to quantify the degree of thoroughness of white-box testing. Coverage testing has the following advantages and disadvantages.

First, reliability seems to increase with test coverage [2]. Malaiya *et al.* [10] presents a logarithmic–exponential model to estimate defect density based on testing coverage measures. Its motivation is to use measurable test coverage to model reliability, because evaluating and projecting reliability growth can help developers or teams optimally allocate resources to meet a deadline with target reliability.

Second, code coverage provides quantification of coverage-related test progress. Selecting tests that provide the largest incremental gain in coverage is one way to prioritize testing. Coverage tools frequently allow detection of redundant test cases, which are candidates for removal from a test suite because executing them takes time and resources without effectively improving defect detection [11].

Third, based on our observations in industry, increasing code coverage becomes a motivation for improving tests. Coverage provides a quantitative measure that can be used to report testing progress. Developers or quality assurance organizations are then more driven to improve the testing process using the guidance of test coverage. We observe developers, and their managers, comparing the coverage they have achieved, competing to see who can get the most coverage earliest.

Because coverage is a quantitative measure, goals for coverage can be established and applied at different testing phases. Developers who strive to achieve the goals, or who are competing to see who can get the highest coverage, are sometimes motivated to run more tests, and gain greater coverage, earlier. We believe that one result is that they find bugs earlier, when the bugs are less expensive to fix. Accordingly, introducing coverage and coverage goals may reduce the cost of correcting errors.

Unfortunately, there is no known underlying theory that predicts how much quality improves with coverage. In practice, very little data or only data for small programs are available that map from coverage measurement to code quality [12]. However, intuitively most developers feel that increasing coverage increases defect detection. Few developers like to deliver code with, say 20% coverage. Unfortunately, we cannot quantify how many more defects are likely to be found by increasing coverage. Full coverage (100%) does not guarantee the absence of defects. There is always a balance between usability and thoroughness when picking a measure. For example, statement coverage is simple but insensitive to conditions. Path coverage is not only too expensive, but lower coverage caused by infeasible paths makes the results hard to explain. If a competition develops based on who can get the most coverage, those whose code checks for unlikely but erroneous conditions, i.e. rare exceptions, may have the hardest time getting high coverage, but may have better quality code.

There is also no good mapping yet known between coverage level and testing effort. We have observed informally that high coverage takes considerable effort and the relation between coverage gained and testing effort seems non-linear. Until more data, and/or better theory, becomes available, we can at best suggest guidelines, such as 70% statement coverage seems achievable with reasonable effort, but it is a struggle to get significantly more than that.

As with other development tools, developers need to learn how to use tools to conduct coverage testing and how to interpret coverage test results. Managers must learn the pitfalls of coverage testing. More machine resources, such as disk space to store coverage reports may also be needed. In other words, just as with other tools, introducing coverage tools requires an investment.

It is easy to be decoyed into focusing on coverage rather than effective testing, if too much emphasis is put on coverage testing and target coverage percent. For example, focusing on writing test cases that cover error-handling code is more difficult [13] and could consume a good portion of developers' time. It is a misuse of coverage testing to write less error-handling code in order to get a better code coverage result.

3. COVERAGE MEASUREMENT

All tools included in this survey have coverage measurement capability. This section compares these tools for three important coverage tool characteristics: (i) supported programming languages, (ii) program instrumentation overhead and (iii) additional features complementary to code coverage.

3.1. Supported languages

Coverage testing tools may apply only to a limited set of programming languages, some to C/C++ only, some to Java only, some to both and some to other languages such as FORTRAN, COBOL or JavaScript. Table 1 shows a complete list of the tools and the languages that they support.

The selection of supported languages reflects each company's target industries. For example, Dynamic Systems focuses on C/C++ because their customers develop real-time systems.

Another very market-conscious decision for C/C++ tool suppliers is the selection of supported platforms, which is not an issue for Java testing tools. Dynamic Memory Systems' customers are mostly medium to large Solaris software development shops [18], thus they only support Solaris. BullseyeCoverage supports a wide range of platforms among code coverage analyzers. Semantic Designs has a general foundation and process for instrumenting

TABLE 1: coverage tools and the languages to which they apply (alphabetical by tool name)

Tool name	C++/C	Java	Other
Agitar [14]		X	
Bullseye [15]	X		
Clover [16]		X	.net
Cobertura [17]		X	
CodeTEST [18]	X		
Dynamic [19]	X		
EMMA [20]		X	
eXVantage [21]	X	X	
Gcov [22]	X		
Intel [23]	X		FORTRAN
JCover [24]		X	
Koalog [25]		X	
Parasoft (C++test) [26]	X		
Parasoft (Jtest) [26]		X	
PurifyPlus [27]	X	X	Basic, .net
Semantic Designs (SD) [28]	X	X	C#, PHP, COBOL, PARLANSE
TCAT [29]	X	X	

source code in not-widely-used languages on a variety of platforms [30].

3.2. Instrumentation overhead

Coverage-testing tools capture coverage information by monitoring program execution. Execution is monitored by inserting probes into the program before or during its execution. A probe is typically a few lines of code that, when executed, generate a record or event that indicates that program execution has passed through the point where the probe is located. There are two kinds of overhead associated with instrumenting a program with probes: the off-line overhead of inserting probes into the program, and the run time overhead of executing the probes to record the execution trace.

3.2.1. Off-line program analysis and instrumentation overhead

Source code instrumentation, used by most of the tools including BullseyeCoverage, Parasoft Insure++, Intel Code Coverage Tool, Semantic Designs and TestWork, requires recompilation, but provides more direct results and is more adaptable to a wide variety of processors and platforms. It cannot be used when the source code is not available, as is often the case for third party code. C/C++ tools such as Dynamic Memory Systems' Dynamics, use runtime instrumentation, which makes them feasible in a production environment. They may be more efficient in terms of compilation time, but less portable. The Java coverage tool Koalog Code Coverage does not require instrumentation, and therefore no recompilation is needed [25]. It operates with the production binaries using the Java Debug Interface, which is part of the Java Platform Debugger Architecture (JPDA). Koalog Code Coverage is platform independent, but requires a JPDA compliant Java Virtual Machine (JVM). Agitar's Agitator runs the code in a modified JVM, also using a dynamic instrumentation approach. eXVantage uses source code instrumentation for C/C++ and bytecode instrumentation for Java. As compared to the other 16 tools, it has the highest off-line instrumentation overhead because it analyzes the program in such a way that it can select the least number of probes to be inserted into the target program.

3.2.2. Run-time instrumentation overhead

Companies that provide tools for system software or embedded software tend to focus more on reducing run-time overhead, so that their tools can be usable in real-time environments, e.g. CodeTEST [18]. TCAT claims that its TCAT C/C++ Version 3.2 maintains its overhead for execution size ratio at 1.1–1.8 and execution speed ratio at 1.1–1.5 [29]; Semantic Designs claims 1.1–1.3, varying according to language and compiler, among the best in our survey. Clover claims that their execution speed overhead is highly variable, depending on the nature of the application under test, and the nature of the tests. Typical execution speed ratio is 1.2–1.5. eXVantage has different versions for different platforms, but claims a ratio of 1.01 for versions optimized real-time, in some environments, based on initial trials (Table 2) [24].

3.3. Additional features

Coverage testing tools can be used to assist in debugging, and some of the coverage tools provide debugging assistance, such as Agitar, Dynamic, JCover, Jtest and Semantic Designs. Each uses a different solution. For example, Agitar provides a snapshot and stack trace to help developers to track the cause of bugs. JCover has the ability to do coverage differencing and comparison to expose the erroneous code. Semantic Designs provides slicing and dicing operations on test coverage data via the GUI to allow code executed/not executed by arbitrary combinations of test runs to be easily isolated visually. eXVantage uses a dynamic execution slicing approach. It creates an execution slice for each test case and reads results from a testing oracle to generate a bug localization report automatically whenever a failed test is detected [8].

Coverage testing tools can also be used for program profiling to identify heavily executed parts of programs. Profiling data can be used in compiler optimization, program refactoring, performance-related debugging, etc. Many tools, including eXVantage, CodeTEST, Dynamic Code Coverage, JCover, PurifyPlus and Semantic Designs, support this feature.

TABLE 2: instrumentation

	Tool name	Source code instrumentation	Byte code instrumentation	On the fly (dynamic)
Java	Agitar [14]			X
	Clover [16]	X		
	Cobertura [17]		X	
	EMMA [20]		X	X
	JCover [24]	X	X	
	Koalog [25]			X
	Jtest [26]		X	X
C/C++	Bullseye [15]	X		
	CodeTEST [18]	X		
	Dynamic [19]			X
	Gcov [22]	X		
	Intel [23]	X		
	C++test [26]	X		
Java and C/C++	eXVantage [21]	X	X	
	PurifyPlus [27]	X	X	
	SD [28]	X		
	TCAT [29]	X		

4. COVERAGE MEASUREMENT CRITERIA

There are a large variety of coverage measurement criteria: data coverage, statement (line) coverage, block coverage, decision (branch) coverage, path coverage, function/method coverage, class coverage and execution state space coverage. Among them, statement (basic block) coverage, decision coverage, function/method coverage and class coverage have been implemented by some coverage tool vendors. The rest remain mostly of interest to researchers, because of their increased complexity and difficulty of use. Practitioners in general do not use other criteria such as data coverage because it is harder to improve data coverage [31]. For example, based on our observations, it is very hard to get better than a few percent data coverage. Software tool companies, especially small software vendors, seek immediate return on their investment, and the level of sophistication of their users directs their focus onto usability, more than thoroughness, or accuracy. Table 3 lists tools with their coverage measurement criteria.

In Table 3, statement coverage means the percentage of (executable) statements executed, while block coverage measures coverage of basic blocks, where a basic block is a sequence of non-branching statements. The results for statement coverage and block coverage could differ, but they are commonly listed in the same category and therefore in the same column in Table 3 for easy comparison. eXVantage and Intel Compiler Code-Coverage measure block coverage. Some of the tools, e.g. Koalog, provide the option of picking the scope for coverage calculations, for example, the statement coverage in a method, a class or a package. Clover allows users to customize the scope by providing sophisticated method- and statement-based filtering of coverage results; this can help to generate more informative reports and will be further discussed in Section 5. Line coverage and statement coverage differ when more than one statement may contribute to a single line's coverage score or one statement takes more than one line. Clover uses statement coverage. However, most of the venders do not distinguish between statement and line coverage, so we list them in the same column in Table 3.

TABLE 3: levels of coverage measurement provided by tools

	Statement/ Line/ Block	Branch/ decision	Method/ function	Class
Agitar	X	X	X	X
Bullseye		X	X	
Clover	X	X	X	X
Cobertura	X	X		
CodeTest	X	X		
Dynamic	X	X	X	
EMMA	X		X	X
eXVantage	X	X	X	
Gcov	X			
Intel	X		X	
JCover	X	X	X	X
Koalog	X		X	
C++test	X	X	X	X
Jtest	X	X	X	X
PurifyPlus	X		X	
SD	X	X	X	X
TCAT	X	X	X	X

There are several variations of condition/decision coverage, but we list all of them in the third column without differentiation. A decision is the whole expression that affects the flow of control in the CFG and is treated as a single node in the CFG. A condition/branch is a sub-expression in a decision expression, connected by logical-and and logical-or operators. BullseyeCoverage and Metrowerk's CodeTEST measure modified condition/decision coverage for C/C++, which provides a good balance of usability and thoroughness [32]. Branch coverage provides the number of branches executed under test. Clover, Cobertura and TCAT/Java support branch coverage for Java.

Method coverage reports for each method or function whether or not it is invoked. Class coverage reports a class as covered if at least one line in that class is executed. Neither method nor class coverage provides fine granularity, but they do provide an overview of testing quality.

Software Research Inc's TCAT/JAVA uses an algorithm, called 'All Paths Generator', which is patented, to calculate simple path coverage. It is intended for use on critical applications where test completeness is required.

The applicability of different measures is affected by the style of the code, such as the size of a method or a function, and the density of branching. For example, method coverage, which counts a method as covered if at least one line in that method is executed, is more suitable for software that consists of many small methods rather than a few large methods.

5. PRIORITIZATION

Formal methods is one approach to produce provably correct software, but in the industrial environment, the use of formal methods has not greatly increased in the past 10 years because of its formidable complexity [33,34]. Instead, testing is generally accepted as standard practice within the industry to improve software quality. More emphasis is put into code coverage measurement when systematic testing is favored. In practice, large complex applications are often tested with low coverage. Based on our industrial observations of the use of eXVantage and other tools, it is easy to get to 65%, however code coverage of 70–80% is often considered acceptable because it is difficult to increase coverage past 80% [35]. Hence, providing assistance in conducting effective testing or prioritizing resources is one of the most important features a good testing tool can provide. An example is identifying critical modules. Among the tools we surveyed, only a handful suggests the availability of such features.

Agitator from Agitar [36] shows risk or complexity scores of classes or methods, which in turn help testers to focus on more complex and therefore presumably more error-prone parts of the code, when time or resources are limited.

Clover provides a sophisticated method- and statement-based filtering of coverage results so that non-critical sections of code can be excluded from coverage calculations. This helps in prioritizing the testing effort. Clover 2 (currently under development) extends this filtering to include cyclomatic complexity [37] and other metrics.

Cobertura [17], a free Java tool that calculates the percentage code accessed by tests, shows the McCabe cyclomatic code complexity of each class, and the average cyclomatic code complexity for each package and for the overall product. Cyclomatic complexity represents the number of paths through a particular section of the code, such as a method in an object-oriented language. It is helpful in pinpointing areas of code that may require additional attention during testing, maintenance or refactoring.

Other tools, such as Dynamic Suite [19], can run in the production mode or even at customer sites to obtain information on which features or modules are being used. The tool has so little performance impact that it can be used in the field during normal system operation to collect operation profiles of the target system without interfering with its normal operation. This operation profile information can guide future testing and therefore help prioritize testing efforts.

eXVantage derives prioritization through enhanced dominator analysis [38,39]. eXVantage prioritization identifies the part of the code that when executed, guarantees that the most code has been executed, i.e. code that can increase coverage the most. As by-products of this dominator analysis, dependency and control flow graphs (CFGs) of source code (when source code is available) are generated. They help visualize the testing coverage.

6. AUTOMATIC TEST GENERATION AND REPORTING

Software testing is a very resource intensive task and automation is one way to decrease the time and cost. Automation of the testing process includes a number of steps, such as test case generation, test execution and creation of test oracles.

The approach to test oracle automation often relies on the system behavior specification and is mostly used in functional testing. Oracles use a system specification to verify the correctness of the target software test results. No full-scale system oracle exists today to achieve this goal. In general, a significant amount of human intervention is still needed to define oracles. Coverage testing is not linked directly to a test oracle, but an automatic test oracle could speed up failed tests detection and testing comprehension.

Automatic test generation remains an important research area, but based on remarks from one vendor, it is overrated as a practical technique, since most tools that automatically generate tests produce tests that cover only trivial boundary condition test cases.

Automated test generation tends to be linked with code coverage, i.e. the goal of generating tests automatically can easily be linked to the goal of increasing coverage. Parasoft C++test generates test cases for C++ programs with good code coverage. Parasoft, Agitar and eXVantage claim the capability of generating Java test cases automatically. Agitator from Agitar provides a certain level of automation by combining test suite generation and execution.

Agitator does Software Agitation, which is defined as 'an automated way of exercising software code and providing observations about its behavior', that is, Agitator creates instances of the classes being exercised, calling each method with sets of input data (provided by static analysis of the source code to cover both boundary conditions and normal conditions), and analyzing the results. Subsequently, a set of summary observations about the behavior is presented to developers who can decide whether the observation is an assertion or a bug. Assertions would be kept for later regression tests or code refactoring.

All the agitation results are stored in XML files, which can be shared among the teams, but would not be useful for other testing tools. The generated tests are not explicitly given to the users. Agitator supports tests created with JUnit by running them as part of each agitation, reporting outcome and coverage. Therefore, test cases that can drive testing efforts independent of any testing tools are not available to the testers.

Besides unit testing, Parasoft provides solutions for web service, functional, rule compliance, security and performance testing. Parasoft Jtest generates test cases in JUnit format to achieve a complete branch coverage. It symbolically executes the code to determine inputs that will cover all the different branches of code. In addition to automatically generating Junit test cases, Jtest can also generate Junit test cases based on monitoring a running Java application, during which users' real inputs will be captured and later used during testing.

eXVantage generates tests to cover high-priority blocks [40]. It includes the following four steps: (i) rank the priority of each target source code line and pick those with the highest priority; (ii) identify paths going through the highest priority points, called hot-spots; (iii) collect and solve constraints on the path and (iv) generate test data to execute the path and render test data into test cases in the same programming language as the original target program.

Besides automation, a user-friendly graphical interface is also an important feature for comparison, since the user interface can be a decisive element in a tool's selection.

Some tools have both a GUI version and a batch mode to suit the requirements of different users. Developers usually like to use the GUI version and the integrators usually like the batch mode version. Java tools, such as Agitar, Clover, Cobertura, eXVantage and Parasoft, include plug-ins for one of the most popular IDE's, Eclipse, which makes the integration in the development stage as transparent as possible. Apache Ant is used as the build tool for some tools, such as Agitar, Clover, Cobertura, Koalog and Parasoft, because it is very commonly used for Java projects.

TABLE 4: tool reporting formats

	GUI	File-based	Notes
Agitar	X		
Bullseye	X	X	CSV, HTML
Clover	X	X	PDF, XML, HTML
Cobertura	X	X	XML
CodeTEST	X	X	
Dynamic		X	
EMMA	X	X	HTML, TXT, XML
eXVantage	X	X	Customizable
gcov		X	
Intel		X	
JCover	X	X	XML, CSV
Koalog	X	X	CSV, LaTex, XML
Parasoft C++test	X	X	Group reporting system
Parasoft Jtest	X	X	Group reporting system
PurifyPlus	X		
SD	X	X	Test coverage vector file, XML
TCAT		X	

One part of the GUI display or the output of the batch mode is the coverage report. Most commercial products include sophisticated report generation components, some of which are graph-based and some file-based. See Table 4 for a list of report formats.

Clover stands out here for integration options. It integrates with all major IDE's (IDEA, Eclipse, JDeveloper, JBuilder, Netbeans) and build tools such as Ant, Maven and legacy shell-script/Make-based systems. Clover also provides historical reporting in HTML or PDF.

Agitar has an innovative way to display coverage reports. Besides coverage information, it uses Agitar Management Dashboard to monitor and manage developer-testing efforts. It is a comprehensive reporting tool, allowing users to input test targets for better management, and providing rule checking functionality.

eXVantage, on the other hand, emphasizes web-based reporting. Its database-reporting feature allows the recording of historic data and scaling up to very large software systems. Historic data allows the observation of trends, which can be used for future predictions of testing.

7. SUMMARY

This survey compares 17 coverage-based testing tools. We also studied other related functionality, which we believe is indispensable for a testing tool if it is to provide an integrated testing solution. Our study includes comparison of three features: (i) code coverage measurement, (ii) coverage criteria and (iii) automation and reporting. Table 5 summarizes our analysis.

TABLE 5: summary table

		Measurements				Reporting		Instrumentation		
Supported languages	Tool name	Statement/ line/block	Branch/ decision	Method/ function	Class	GUI	File-based	Source code instrumentation	Byte code instrumentation	On the fly (dynamic)
Java	Agitar [14]	X	X	X	X	X				X
	Clover [16]	X	X	X	X	X	X	X		
	Cobertura [17]	X	X			X	X		X	
	EMMA [20]	X		X	X	X	X		X	X
	JCover [24]	X	X	X	X	X	X	X	X	
	Koalog [25]	X								X
	Jtest [26]	X	X	X	X	X	X	X	X	X
C/C++	Bullseye [15]		X	X		X	X	X		
	CodeTEST [18]	X	X			X	X	X		
	Dynamic [19]	X	X	X			X			X
	Gcov [22]	X					X	X		
	Intel [23]	X		X			X	X		
	C++test [26]	X	X		X	X	X	X		
Java and	eXVantage [21]	X	X	X	X	X	X	X	X	
C/C++	PurifyPlus [27]	X		X		X		X	X	
	SD [28]	X	X	X	X	X	X	X		
	TCAT [29]	X	X	X	X		X	X		

Besides C/C++ and Java, the following tools also support: Clover supports .net; Intel supports FORTRAN; PurifyPlus supports Basic and .net; SD supports C#, PHP, COBOL and PARLANSE.

Each tool has its pros and cons depending on its application domain(s). For example, Semantic Designs' differentiator is their parsing capability for various languages, including some often considered obsolete. The strong point of Agitar is mutation-based data input to achieve very high code coverage running on their provided platform. Dynamic Memory owns unique dynamic instrumentation technique, which provides excellent tool usability. The eXVantage tool suite, on the other hand, differentiates from other tools mostly in automatic unit test generation and low overhead. It can automatically generate test cases to reach high-priority points in the program. Table 6 provides guidelines for developers to select coverage testing tools.

Overall, much research in the area of software coverage testing has been realized and used in industrial software production. We hope our work will contribute to more usage of tools to improve software testing.

TABLE 6: tool selection guideline

Need	Tool(s)
Real-time/low overhead	Dynamic, eXVantage
High coverage	Agitar, Parasoft Jtest
Multi-language support	PurifyPlus, Semantic Designs
Multi-platform (C++ only)	BullseyeCoverage, Semantic Designs

ACKNOWLEDGEMENTS

We thank all the people who carefully read the preliminary versions of this paper and helped to improve it. In particular, Matthew Fisher from Dynamic Memory Solutions, Erika Delgado from Parasoft, Ira Baxter from Semantic Designs and Brendan Humphreys from Clover Support reviewed the initial version of this paper and provided very valuable comments. Steve Cornett from BullseyeCoverage reviewed the initial version and responded to our questionnaire. For their technical help, we are grateful to our colleagues Jon Bentley at Avaya Labs Research and Tom Walsh. We also would like to thank the reviewers for their detailed and helpful comments on an earlier version of this article.

REFERENCES

[1] Frederick, P. and Brooks, J. (1995) *The Mythical Man-Month* (anniversary edn). Addison-Wesley, Boston, USA.

[2] Yang, M.C.K. and Chao, A. (1995) Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs. *IEEE Trans. Reliab.*, **44**, 315–321.

[3] Grindal, M., Offutt, J. and Melin, J. (2006) On the Testing Maturity of Software Producing Organizations. *Proc. Testing: Academic and Industrial Conference On Practice And Research Techniques*, Windsor, DC, UK, August 29–31, pp. 171–180. IEEE Computer Society, Washing, USA.

[4] Zhu, H., Hall, P.A.V. and May, J.H.R. (1997) Software unit test coverage and adequacy. *ACM Comput. Surv.*, **29**, 366–427.

[5] Weiss, D., Bennett, D., Payseur, J., Zhang, P. and Tendick, P. (2002) Goal-Oriented Software Assessment. *Proc. 24th International Conference on Software Engineering*, Orlando, FL, USA, May 19–25, pp. 221–231. ACM Press, New York, NY, USA.

[6] Repenning, N.P., Gonçalves, P. and Black, L.J. (2001) *Past the Tipping Point: The Persistence of Firefighting in Product Development.* Cambridge, MA, USA 02142. http://web.mit.edu/nelsonr/www/TippingV2_0-sub_doc.pdf.

[7] Maranzano, J., Rozsypal, S., Warnken, G., Weiss, D., Wirth, P. and Zimmerman, A. (2005) Architecture reviews: practice and experience. *IEEE Softw.*, **22**, 34–43.

[8] Wong, W.E. and Li, J. (2005) An Integrated Solution for Testing and Analyzing Java Applications in an Industrial Setting, *Proc. 12th Asia-Pacific Software Engineering Conference (APSEC'05)*. Vol. 00, 576–583. IEEE Computer Society.

[9] Stokes, D.E. (1997) *Pasteur's Quadrant, Basic Science and Technological Innovation.* Brookings Institution Press. Washington, DC, USA.

[10] Malaiya, Y.K., Li, M.N., Bieman, J.M. and Karcich, R. (2002) Software reliability growth with test coverage. *IEEE Trans. Reliab.*, **51**, 420–426.

[11] Jones, J.A. and Harrold, M.J. (2003) Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, **29**, 195–209.

[12] Xia, C. and Michael, R.L. (2005) The Effect of Code Coverage on Fault Detection under Different Testing Profiles. *Proc. First International Workshop on Advances in Model-Based Testing*. ACM Press, St. Louis, MO.

[13] Fu, C. and Ryder, B.G. (2007) Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. *Proc. 29th International Conference on Software Engineering*, ACM Press, Minneapolis, MN, USA.

[14] Agitar, http://www.agitar.com/.

[15] BullseyeCoverage, http://www.bullseye.com/index.html.

[16] Clover, http://www.cenqua.com/clover.

[17] Cobertura, http://cobertura.sourceforge.net/.

[18] CodeTest, http://www.metrowerks.com/MW/Develop/AMC/CodeTEST/default.htm.

[19] Dynamic Memory Systems, http://dynamic-memory.com/.

[20] EMMA, http://emma.sourceforge.net/.

[21] eXVantage, http://www.research.avayalabs.com/user/jjli/eXVantage.htm.

[22] gcov, http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[23] Intel code coverage tool, http://www.intel.com/cd/software/products/asmo-na/eng/219794.htm.

[24] JCover, http://www.mmsindia.com/JCover.html.

[25] Koalog, http://www.koalog.com/php/kover.php.

[26] Parasoft, Jtest: http://www.parasoft.com/jsp/products/ home.jsp?product=Jtest Parasoft C++test: http://www.parasoft.com/jsp/products/home.jsp?product=CppTesthttp://www.parasoft.com.

[27] Purify, Plus, http://www-306.ibm.com/software/awdtools/purifyplus/.

[28] Semantic, Designs, http://www.semdesigns.com/index.html.

[29] TCAT, http://www.soft.com/TestWorks/.

[30] Baxter, I.D. (2002) *Branch Coverage for Arbitrary Languages Made Easy.* http://www.semdesigns.com/Company/Publications/TestCoverage.pdf

[31] Weyuker, E.J. (1984) The complexity of data flow criteria for test data selection. *Inf. Process. Lett.*, **19**, 103–109.

[32] Chilenski, J.J. and Miller, S.P. (1994) Applicability of modified condition/decision coverage to software testing. *Softw. Eng. J.*, **9**, 193.

[33] Bowen, J.P. and Hinchey, M.G. (2005) Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. *Proc. 10th International Workshop on Formal Methods for Industrial Critical Systems*. ACM Press, Lisbon, Portugal.

[34] Bowen, J.P. and Hinchey, M.G. (1995) Ten commandments of formal methods. *Computer*, **28**, 56–63.

[35] Piwowarski, P., Ohba, M. and Caruso, J. (1993) Coverage Measurement Experience During Function Test. *Proc. 15th International Conference on Software Engineering*. IEEE Computer Society Press, Baltimore, MD, USA.

[36] Boshernitsan, M., Doong, R. and Savoia, A. (2006) From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. *Proc. ISSTA '06*. ACM Press, Portland, ME, USA.

[37] McCabe, T. (1976) A complexity measure. *IEEE Trans. Softw. Eng.*, **5**, 45–50.

[38] Agrawal, H. (1994) Dominators, Super Blocks, and Program Coverage. *Proc. 21st ACM Sigplan-Sigact Symposium on Principles of Programming Languages*. ACM Press, Portland, OR, USA.

[39] Li, J.J. (2005) Prioritize Code for Testing to Improve Code Coverage of Complex Software. *Proc. 16th IEEE International Symposium on Software Reliability Engineering*, IEEE Computer Society.

[40] Li, J.J., Weiss, D. and Yee, H. (2006) Code-coverage guided prioritized test generation. *Inf. Softw. Technol.*, **48**, 1187–1198.

APPENDIX

Note sent to tool vendors

The information we collected is mostly from your website. Please let us know if you have any updates or corrections. In addition, if possible, we would like to know the extent of your user base, the instrumentation overhead of your tool at run time (if you have such data), license cost and any other information that you think might be valuable for our study.