CHALMERS

Code coverage criteria and their effect on test suite qualities

Master of Science Thesis in the Programme Software Engineering and Technology

MIKHAIL KALKOV

DZMITRY PAMAKHA

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, December 2013

Code coverage criteria and their effect on test suite qualities

MIKHAIL KALKOV
DZMITRY PAMAKHA

Examiner: MIROSLAW STARON

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden December 2013

Abstract

The efficiency of software testing is often assessed based on the results of code coverage analysis, which is performed using such criteria as function, statement and condition/decision coverage. However, it is not uncommon for a criterion definition to vary between tool vendors and programming languages, which makes any analysis and comparison difficult. Given this uncertainty, one also wonders what evidence exists to support the use of code coverage data as a basis for evaluating test efficiency. In this paper various coverage criteria are analyzed and the reason the majority of them are language-specific is discussed. Following that, a test suite quality model is proposed and used to demonstrate the non-linear relationship between defect detection and coverage levels as well as how code coverage data can be used to minimize a test suite, improve its regression sensitivity or even generate missing test cases. Finally, several coverage analysis tool case studies are done and an implementation of a BullseyeCoverage plug-in for Eclipse C/C++ Development Tools environment is described. This report is concluded with a discussion of open questions, such as the feasibility of precisely specified universal coverage criteria, automated coverage-guided test case generation and an approach to measure coverage of generated source code.

Table of Contents

1. Introduction to code coverage

Software has become a huge industry worldwide since the invention of the first general-purpose programmable electronic computers in the mid-20th century. According to a report by Forrester Research (2011, cited in Dignan 2011), global software spending totaled $402 billion in 2010, and surpassed computer equipment spending, which amounted to $337 billion. Furthermore, software plays an especially important role for Swedish industry. For example, 80% of Ericsson R&D costs as well as 70% of Volvo trucks innovations are in software (Swedsoft 2010).

Despite the prevalence of software, its quality often leaves much to be desired. According to a planning report commissioned by the US National Institute of Standards and Technology, inadequacy of software testing infrastructure cost the US economy $59.5 billion in 2002, $21.2 billion of which were spent by the developers, and the rest by the users (RTI International 2002, p. ES-11). Therefore, improvements in software testing could provide extensive savings for both parties.

In order to assure quality of a modern software system, which may consist of thousands of modules and millions of source code lines, one has to apply various techniques throughout software development life-cycle. Overall quality goals are set already during requirements analysis. During design the system is split into components with well-defined responsibilities, interfaces and quality requirements. Each component is then implemented and tested separately before the system is assembled and tested as a whole. Finally, the post-release quality of software is tracked and necessary adjustments are implemented.

The general goal of testing in this process is to verify that software meets its functional and non-functional requirements and has no unintended functions. When testing safety-critical systems, one starts with requirements coverage analysis in order to verify that all requirements have corresponding tests. Thereafter, structural coverage analysis is undertaken in order to verify absence of code segments which are *not* exercised by requirements-based tests and therefore may implement unintended functions. Thus, correspondence between requirements and actual software is established. (Hayhurst – Veerhusen – Chilenski – Rierson 2001, p. 4)

For many non-safety-critical systems, however, requirements specification may lack rigor or be not up-to-date, which complicates creation of a proper test suite and identification of code segments that may implement unintended functions. In this case the goal of testing becomes to verify that software meets existing requirements, and requirements coverage analysis is skipped. Furthermore, structural coverage analysis is performed in order to identify code segments which are not exercised by existing tests and therefore may prompt creation of additional test cases (Myers 2004b). In such situations the term “structural coverage” is often shortened to “test coverage” or “code coverage”.

Regardless of the approach to testing, structural coverage is usually expressed in the form of source code annotations or summarized as a percentage of specified code structure elements covered by tests from the total number of such elements. Some of widely used code structure elements are

source code lines, function entry points, statements and decision points. Although code coverage measurements are often used to assess completeness of a test suite, it is possible to utilize this data for different purposes. For example it can be used to minimize the test suite execution time without compromizing its completeness or to maximize the probability that regression errors are detected by one of the first test cases.

Unfortunately, there is not much research either into ways to utilize code coverage data or into ways to present it. This can probably be attributed to the fact that there is no common vocabulary of code structure elements and that there exist contradicting definitions for widespread coverage criteria. So in practice the semantics of code coverage data depends on the programming language and the used coverage tool. (Holzer – Schallhart – Tautschnig – Veith 2009, 2)

2. Research purpose and methods

2.1 Background

The authors of this paper were given a task to implement a plug-in for Eclipse IDE, which would incorporate code coverage data produced by BullseyeCoverage software into Eclipse C/C++ Development Tools environment. This plugin was to be made available for embedded developers at one department of a large Swedish telecommunications company, who used to get code coverage analysis results printed out in plain text to a black-and-white terminal window, often thousands lines at a time. The main goal was to give software developers a quick and easy to use access to code coverage data so that insufficiently tested components are discovered earlier in the software development life-cycle and only adequately tested software is sent further to the Integration and Verification department.

During the preliminary investigation we attempted to find out how code coverage data can be utilized by a software developer. Surprisingly we were unable to locate a holistic analysis of code coverage utilities, although many papers touched different aspects of this question. Similarly, we have been unable to find a study of various means to present code coverage data. A quick review of several code coverage tools has shown that most of them utilize code coverage data only to find insufficiently tested parts of source code and visualize this information in unique ways. After realizing that definitions of coverage criteria vary between tools, we have settled to investigate the following questions.

2.2 Research questions and limitations

Question 1: What are common coverage criteria and their limits of applicability?
We intend to describe coverage criteria widely known in academic world as well as in industry and compare them to each other. We would also analyze if these criteria are applicable to several programming languages, which represent major programming paradigms.

Question 2: Which test suite qualities can be measured or affected by utilizing code coverage data?
Coverage analysis is often employed to measure test suite completeness, and we wonder if there is research supporting this practice. More generally, we would like to know which other test suite qualities can be measured or affected through the use of code coverage data.

Question 3: What are common ways to present code coverage data and do they faithfully reveal important test suite qualities?
There are numerous tools for code coverage analysis, and multiple ways to present coverage data. We wonder which test suite qualities are revealed by common presentation formats.

Question 4: How to implement an Eclipse plug-in for BullseyeCoverage so that code coverage data can be fully utilized?

We conclude this report by discussing an actual implementation of the Eclipse plug-in for BullseyeCoverage.

2.3 Research methods

Answers for the first three questions were sought by reviewing literature and performing case studies. In particular ACM and IEEE Xplore digital libraries along with the Google web and Scholar search engines were used to find scientific papers which match a combination of such keywords as “test coverage”, “code coverage”, “test suite”, “test effectiveness”, “software quality”, “software verification”, “post-release defects”, as well as the names of specific coverage criteria, test suite qualities and coverage analysis tools. The discovered papers were reviewed, and the ones, which contained results that were novel at the time of publication and are relevant to the research questions, have been used. In addition to search engines, the “Cited by” functionality of digital libraries have been used on some works to discover more recent studies on the same subject.

In the course of coverage criteria research, several coverage tools were discovered through references to them in scientific papers, whereas other tools were discovered through exploration of English-language Wikipedia articles related to code coverage. In the end, Google search engine was used to identify miscellaneous code coverage tools and assess their popularity based on their position in the list of returned results.

We have also proposed to use test case and test suite qualities in order to be able to analyze relationship of code coverage and testing efforts. This idea has been inspired by “Characteristics of a Good Requirement” and “Characteristics of a Good Requirements Specification” discussed in the book Software & System Requirements Engineering In Practice (Berenbach – Paulish – Kazmeier – Rudorfer 2009, pp. 9-15), which in turn refers to IEEE Standard 830 “Recommended Practice for Software Requirements Specifications” from 1998. The lists of test case and test suite qualities have been put forward using brainstorming, and research literature has been reviewed in search of proofs of relationship between code coverage and each such quality.

Four different coverage tools have been selected for a case study so that they represent the two most widespread programming platforms — C/C++ and Java — which have been used to develop software ranging from embedded to desktop and server applications. For each tool, its web site and documentation has been studied, and if possible a test installation was done in order to perform first-hand evaluation.

Finally, in order to answer the last question, an implementation of a code coverage plug-in for Eclipse IDE has been attempted. Based on answers for previous questions, several features have been proposed for an industry customer. The work has been performed in an agile manner with two-week iterations, and in the beginning of each iteration, the customer representative helped us prioritize work for the upcoming two weeks. Soon after the first prototype was ready, authors of this paper have been offered employment by the customer, and upon accepting the offer, we have been given unrelated tasks. So, the plug-in prototype was put in production and the advanced features we planned were never implemented.

3. Coverage criteria and their limits of applicability

In order to measure code coverage, coverage criteria have to be selected. There exist multiple alternatives, and many tools support more than a single one, which makes it possible to analyze coverage simultaneously from several viewpoints. However, the precise definition of each criterion varies from one tool vendor to another one and from one programming language to another. Furthermore, conceptually similar criteria may have unrelated names, which makes describing and analyzing them difficult.

This chapter starts by introducing common notation and describing the following coverage criteria: function, loop, line, statement, block, decision, condition, condition/decision, multiple condition, modified condition/decision, linear code sequence and jump, path, and synchronization coverage. It continues by comparing these criteria to each other and concludes by analyzing their applicability to software written in diverse programming languages.

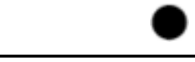
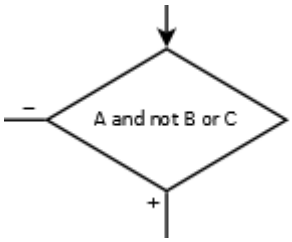
3.1 Notation description and a sample program

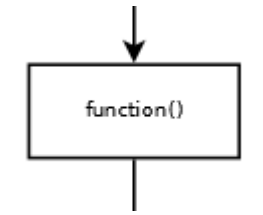
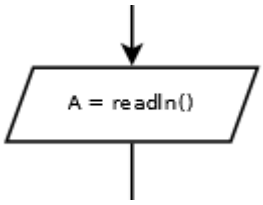
To demonstrate various coverage criteria, a sample program will be used. It will take two numbers and print the former raised to the power of the latter. Furthermore, in order to keep the discussion clear of language-specific details, this program will be written in a simple programming language which has the following features:

- built-in functions such as string concatenation or Boolean logic operators,
- support for user-defined functions,
- support for conditional execution of one out of two branches,
- support for loops, and
- support for synchronous input and output as well as variables.

The subsequent flowchart notation will be used to write programs in this language.

Table 1. Flowchart notation

Symbol	Explanation
	an entry or exit point of a function
	the flow of control
	a branching point, where expression in the rhombus represents a condition, “+” marks the branch executed if condition was True “–” marks the branch executed if condition was False

	a function call. In this example the return value of a function is discarded.
	an input or output function. In this example the value returned by an input function is assigned to variable A.

The sample program consists of three user-defined functions: main(), isValid(Value, Value) and power(Number, Integer). “main” function represents the whole program, “isValid” decides if input is valid or not, and “power” performs the calculation. The functions are given below.

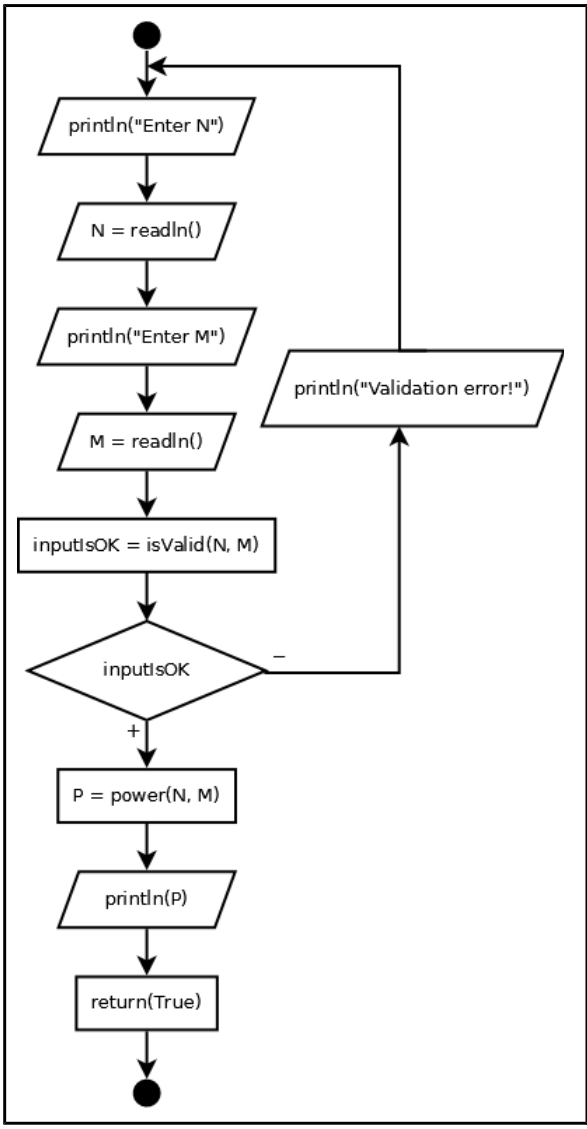


Figure 1. main() function

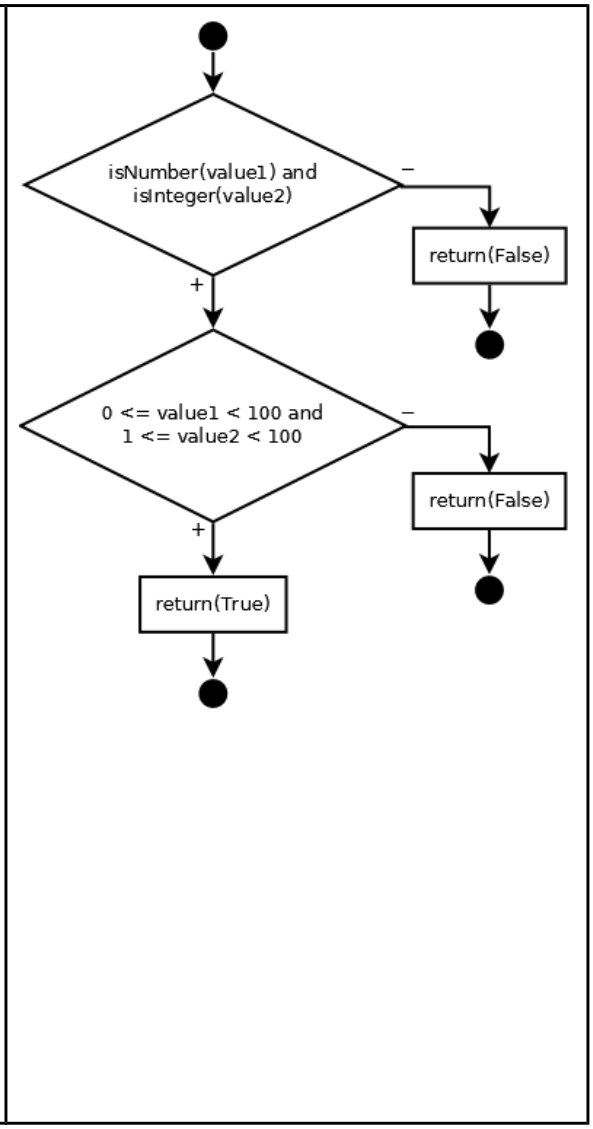


Figure 2. isValid(value1, value2) function

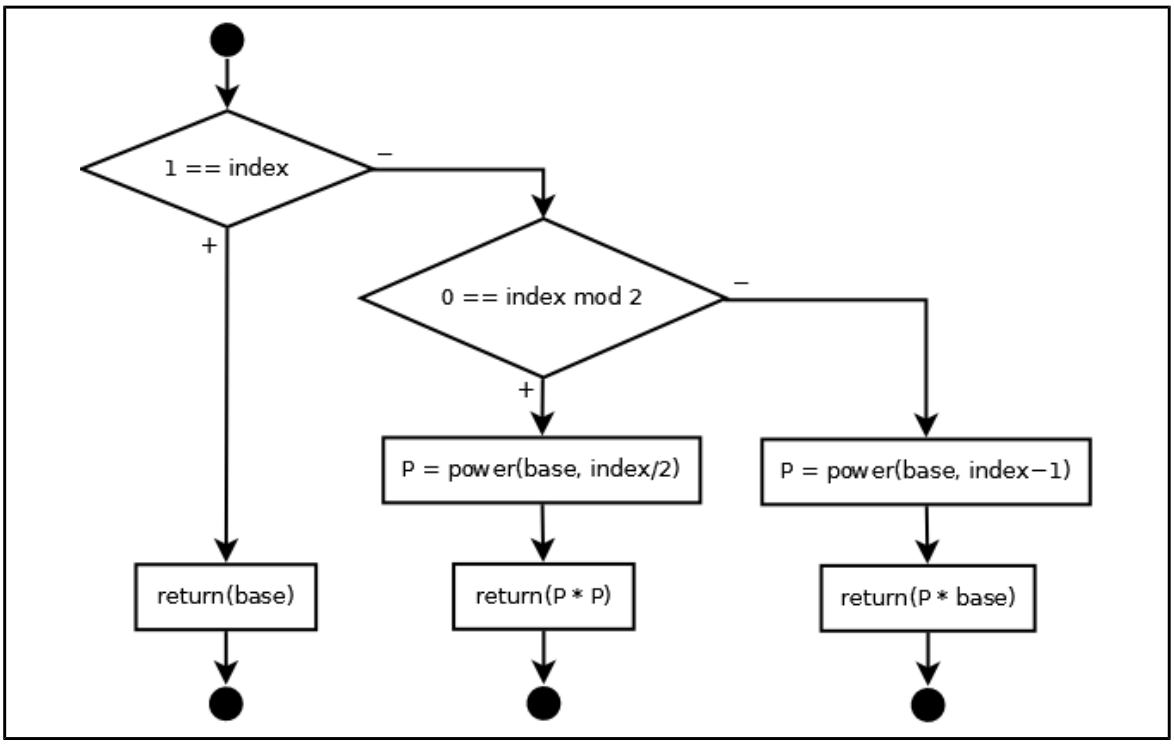


Figure 3. power(base, index) function

3.2 Coverage criteria

3.2.1 General considerations

Code coverage is meaningless when the boundaries of software under test are not stated. In other words, one has to decide if he or she is going to measure coverage for a single source file, a library, an application, an operating system, etc.

Furthermore, it is important to decide if one measures the coverage of source or object code since they are not necessarily equal (Hayhurst – Veerhusen – Chilenski – Rierson 2001, p. 11). For example, a single statement in source code can be represented by a sequence of statements in object code, and the control flow graph can be modified by the compiler during optimization.

Finally, the discussion below focuses on few popular control flow coverage metrics leaving alone less popular control flow and all data flow ones. An informal overview of many of them can be found in Steve Cornett's article (Cornett 2011).

*3.2.2 Function coverage**

This criterion is used to assess the proportion of functions called during testing from the total number of functions in the program, and there exist several variations of it. For example,

---

**Function coverage should not be confused with functional coverage, which is a term for requirements coverage analysis in the integrated circuit design community (Spear – Tumbush 2012).*

BullseyeCoverage only measures if a function was entered at least once (Cornett 2011), whereas gcov reports the number of times a function was entered and exited (Free Software Foundation 2010a). Hereafter it is assumed that a function is covered if it is entered at least once but not necessarily executed until an exit point.

For example, here is what function coverage of the sample program would be if the measurements are done for the three aforementioned functions and if isNumber(value) and isInteger(value) built-in functions only accept natural numbers in decimal notation.

Table 2. Function coverage of the sample program

Input	Function Coverage
12	Functions: 1. main is covered 2. isValid is uncovered 3. power is uncovered Total: 33% (1 out of 3 functions covered)
asdf qwer zxcv zxcv	Functions: 1. main is covered 2. isValid is covered 3. power is uncovered Total: 67% (2 out of 3 functions covered)
123 32 12 32	Functions: 1. main is covered 2. isValid is covered 3. power is covered Total: 100% (3 out of 3 functions covered)

3.2.3 Loop coverage

Many programming languages have special constructs to let programmer iterate over several values in a group, and a flowchart representation of such constructs contains a closed control flow loop. Depending on input, the loop body may be skipped, executed once or several times, and it is beneficial to have at least one test case for each of these boundary conditions (Myers 2004a). Loop coverage measures whether each loop body was executed zero, one or more times (Cornett 2011) and thus reveals which typical loop boundary conditions have been tested.

The main() function of the sample program contains one loop, which is covered by different test cases according to the table below.

Table 3. Loop coverage of main() function

Input	Loop coverage
a	Loop 1: uncovered

2 3	Loop 1: one iteration
a b 2 3	Loop 1: more than one iteration

3.2.4 Line, statement and block coverage

These three criteria assume that a program is a series of code units executed sequentially but disagree what constitutes a single code unit. *Line coverage* considers a code unit to be any sequence of expressions stretching a single line of source file. According to *statement coverage*, a code unit is a single statement regardless of the number of lines it occupies. Finally, *block coverage* assumes that a unit of code is a sequence of non-branching expressions (Koskela 2004). Whenever a language grammar makes a distinction between an expression and a statement the precise meaning of these criteria has to be further investigated.

Let us consider what line, statement and block coverage would be for the main() function of the sample program if it was written in pseudo code as given below. In order to simplify analysis, each line is annotated with a comment explaining what it contains.

Table 4. Decomposition of main() function's pseudo code into units

Row #	Pseudo code	Expressions
1	function main()	none
2	begin loop	none
3	println(“Enter N”)	function call
4	N = readln()	function call and variable assignment
5	println(“Enter M”)	function call
6	M = readln()	function call and variable assignment
7	inputIsOK = isValid(N,M)	function call and variable assignment
8	exit when inputIsOK	condition evaluation
9	println(“Validation error!”)	function call
10	end loop	none
11	P = power(N,M)	function call and variable assignment
12	println(P)	function call
13	return(True)	function call
14	end function	none

From the table above it follows that main() function contains 14 expressions, and 10 lines with at least one expression. Additionally, there are three blocks: lines 3 through 8, line 9 and lines 11 through 13. Thereby, the coverage of the main() function would look as follows with different input values.

Table 5. Line, statement and block coverage of main() function

Input	Line coverage	Statement coverage	Block coverage
a	3 out of 10 (30%)	4 out of 14 (29%)	0 out of 3 (0%)
a b	7 out of 10 (70%)	10 out of 14 (71%)	2 out of 3 (67%)
1 2	9 out of 10 (90%)	13 out of 14 (93%)	2 out of 3 (67%)
a 2 3 4	10 out of 10 (100%)	14 out of 14 (100%)	3 out of 3 (100%)

3.2.5 Decision, condition, condition/decision (C/D) and multiple condition (MCC) coverage

Unlike statement coverage, which analyzes sequential statements, *decision coverage* measures whether all branches were exercised at each decision point. An alternative to decision coverage is *condition coverage*, which shows if all atomic conditions in each decision point were evaluated both to True and to False. Finally, *condition/decision coverage* (often abbreviated to "C/D") is simply a combination of decision and condition metrics. (Myers 2004a) According to DO-178B guidelines, these criteria additionally require every entry and exit point in the program to be invoked at least once (Hayhurst – Veerhusen – Chilenski – Rierson 2001, p. 7), but this is not assumed herein.

Consider the execution of isValid(value1, value2) function with the following input data assuming that condition evaluation is not short-circuited.

Table 6. Decision, condition and C/D coverage of isValid(value1, value2) function

Input	Decision coverage	Condition coverage	C/D coverage
1 2	2 out of 4 (50%)	6 out of 12 (50%)	8 out of 16 (50%)
1 2 a b	3 out of 4 (75%)	8 out of 12 (67%)	11 out of 16 (69%)
1 2 a b 1 101	4 out of 4 (100%)	9 out of 12 (75%)	13 out of 16 (81%)
1 2 a b -1 -1 101 101	4 out of 4 (100%)	12 out of 12 (100%)	16 out of 16 (100%)

To better understand the difference between these metrics, let us examine one conditional expression from this function in more detail.

1. "`isNumber(value1) and isInteger(value2)`" is a compound condition, the value of which becomes a decision.
2. "`isNumber(value1)`" is an atomic condition
3. "`isInteger(value2)`" is another atomic condition which is joined with the previous one by the logical AND to produce a compound condition.

Each of these parts can be in one of the four states: uncovered, evaluated to True, evaluated to False and evaluated to both True and False. Let us see how decision, condition and condition/decision coverage compare to each other.

Table 7. Computation of decision, condition and C/D coverage for a single condition

Input	`isNumber (value1)`	`isInteger (value2)`	`isNumber(value1) and isInteger(value2)`	Decis. cov.	Cond. cov.	C/D cov.
1 2 a b	True False	True False	True False	100%	100%	100%
1 a a 2	True False	False True	False False	50%	100%	83%
1 2 a 2	True False	True True	True False	100%	75%	83%

As can be observed from the table above, there are corner cases when coverage reported according to decision or condition metrics might be misleading, and this is why condition/decision coverage is often used instead of them. However, there is one more alternative for condition coverage called *multiple condition coverage* (abbreviated to "MCC"), which requires every possible combination of conditions to be tested (Myers 2004a). For example, to achieve 100% MCC at a decision point with two atomic conditions they have to take these values: (*False*, *False*), (*False*, *True*), (*True*, *False*), (*True*, *True*). In general MCC requires at least 2^n tests for a decision point with n atomic conditions. Thus, it overcomes shortcomings of decision and condition coverage criteria but leads to exponential explosion in the number of required test cases, and therefore is less useful.

3.2.6 Modified condition/decision coverage (MC/DC)

MC/DC incorporates function coverage and extends condition/decision coverage with a requirement that every condition in a decision has to affect that decision's outcome independently from other conditions (Chilenski – Miller 1994). This metric is widely used in safety-critical systems and is even mandated by the US Federal Aviation Administration for level A airborne computer software through its acceptance of DO-178 guidelines (Hayhurst – Veerhusen – Chilenski – Rierson 2001, p. 1), which means MC/DC is well-defined for imperative languages. There are certain subtleties in the definition of MC/DC and the first one concerns the exact definition of a decision. According to MC/DC, a decision is any conditional expression, which may occur within a branch point or outside of it, e.g. in a variable assignment (ibid., p. 13). Other confounding issues are how condition

independence can be demonstrated and how coupled conditions such as several occurrences of the same variable affect MC/DC calculation.

The original "unique-cause" approach to these problems requires one to vary each condition while holding other conditions fixed, and therefore precludes the possibility of reaching full MC/DC coverage for a decision with coupled variables. In general it requires between $n+1$ and $2n$ test cases for a decision with n conditions. (Chilenski – Miller 1994, p. 197)

A more recent masking approach to MC/DC relies on analysis of Boolean logic and thereby vastly decreases the number of Boolean expressions for which full MC/DC coverage cannot be reached (Chilenski 2001, p. 59). This approach also decreases the minimum number of test cases to $\lceil\sqrt{2n}\rceil$ compared to unique-cause. (ibid, p. 31).

For example, expression $(A \wedge B) \vee (\neg A \wedge C)$ cannot be fully covered according to unique-cause MC/DC because it is not possible to vary A_1, while holding B, C and A_2 fixed. On the other hand, this expression can be fully covered with the four test cases in the table below if one employs masking MC/DC.

Table 8. Test cases providing masking MC/DC for $(A \wedge B) \vee (\neg A \wedge C)$

Test Case #	A	B	C	$(A \wedge B) \vee (\neg A \wedge C)$
1	False	False	True	True
2	False	True	False	False
3	True	False	True	False
4	True	True	False	True

Either test cases 1 and 3 or test cases 2 and 4 show that the value of A affects the decision outcome independently. Test cases 1 and 2 demonstrate this is true for C because $A = False$ masks the value of B and the whole first term evaluates to *False* so the value of the second term becomes the decision outcome. Similarly, test cases 3 and 4 demonstrate independence of B.

In order to compare C/D coverage with MC/DC another sample program will be used since cannot be demonstrated with the one used above. Let OR operator evaluate its left operand first and mark input values, which affect the decision independently from others, with **bold typeface**.

Table 9. Difference between C/D coverage and MC/DC

Input A	Input B	$A \vee B$	C/D coverage	MC/DC
False **True**	**False** True	False True	6/6 = 100%	5/6 ≈ 83%
False **True** False	**False** False **True**	False True True	6/6 = 100%	6/6 = 100%

In the first case MC/DC was not complete because the value of input B was masked by $A=True$ and did not affect the outcome of decision $A \vee B$. Addition of the third test case has solved this problem.

It is worth mentioning that modified condition/decision coverage is not implied by multiple condition coverage (MCC) described in the previous section because there exist boolean expressions for which full MC/DC cannot be obtained. Consider the following C program.

```
if (A || !A)
     statement1;
else
     statement2;
```

Listing 1. Code sample demonstrating the difference between MCC and MC/DC

Since MCC does not require the absence of tautologies in decision predicates, a 100% MCC level is feasible, whereas the highest level of MC/DC is 50% in this case.

3.2.7 Linear code sequence and jump (LCSAJ)

When Michael Hennel, Martin Woodward and David Hedley proposed LCSAJ in their seminal work "On program analysis" (Hennel – Woodward – Hedley 1976), they defined it in terms of a low-level programming language which consists of linear instruction sequences as well as conditional and unconditional jumps. According to Hennel et al., the start point of a linear code sequence is "*either the first line of the program or any line which has a jump to it from elsewhere in the program, other than the preceding line*", and the end point is "*either the end of the program or a line which contains a jump to other than the succeeding line*". Nevertheless, LCSAJ was successfully mapped onto some high-level languages such as Algol 68, and Fortran IV (ibid.). In case of Fortran, the following start points are listed by Hennel et al.: "the start of the program, any label which is accessed from other than the preceding line, the start of a DO loop, the start of any subroutine or function, or any line which contains a subroutine or function call", whereas the end points could be represented by "a program stop, any GOTO, assigned GOTO, computed GOTO or arithmetic IF, which contains a jump to a label which is not on the next line, the return of any subroutine or function, or any line which contains a subroutine or function call".

It is not immediately clear if LCSAJ can be applied to programs written in the simple programming language described in the beginning of this chapter. This language lacks support for unstructured control flow possible with *goto* statements and continuations but includes constructs for user-defined functions, conditional execution and loops. At this point one could suggest to apply the Fortran IV rules described above, but they rely on the fundamental assumption that a program is uniquely translated into a sequence of linear commands intermingled with conditional and unconditional jumps, and there is no such unique translation for a program written in our language, which is defined in terms of a control flow graph. What is more, even if one produces a set of rules, which describes such a translation for the abstract language used above, it will conflict with existing programming language implementations, which may apply different translation rules.

One potentially conflicting implementation is that of C language compilers. Since the C language standards do not define exactly how source code should be translated into object code consisting of linear code sequences and jumps, compiler implementations may arbitrarily reorder object code instructions (Free Software Foundation, 2010b.). The following figure demonstrates how the C language if-then-else statement might look like when translated to a low-level assembly language. The actual translation depends on the compiler and its configuration, and it is conceivable that the order of branches might be swapped or one of the branches could be moved to a separate memory location thereby altering jump locations.

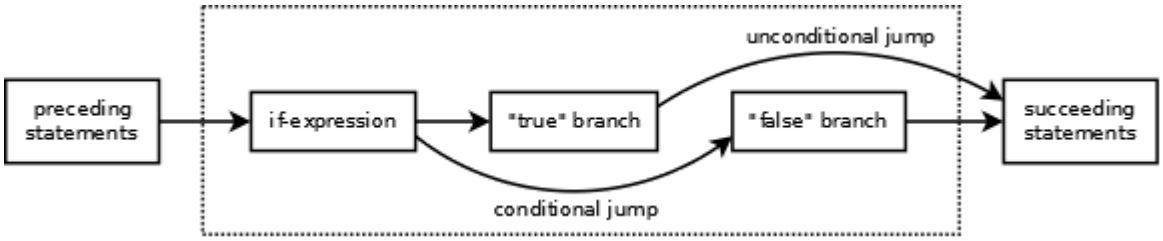


Figure 4. A low-level language equivalent to the C language if-statement

However, assuming the above low-level representation of an if-statement, the following LCSAJ would cover this code:

1. start in the preceding statements, end at the if-expression, and jump to the false branch
2. start in the preceding statements, end after the "true" branch, and jump to the succeeding statements
3. start at the "false" branch, and continue with the succeeding statements

Next, assuming that pre- and post-condition loops are translated in a certain way by the C compiler, the following two figures demonstrate which LCSAJ sequences might be needed to cover them.

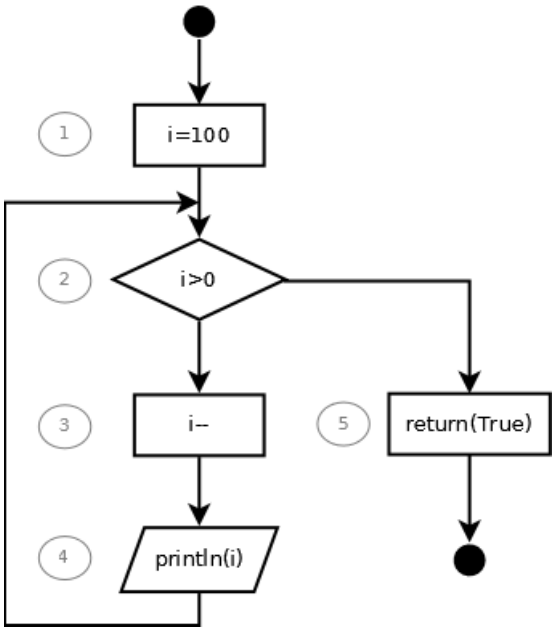


Figure 5. A precondition loop

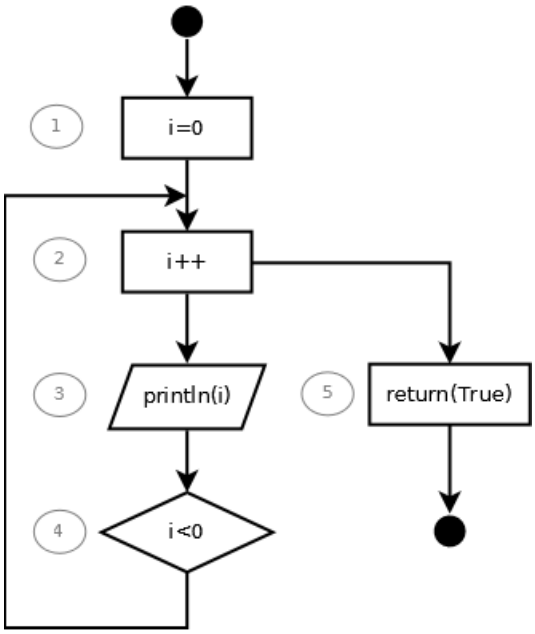


Figure 6. A postcondition loop

Table 10. LCSAJs for a precondition loop

Start at	1	1	2	2	5
End at	2	4	2	4	5
Jump to	5	2	5	2	…

Table 11. LCSAJs for a postcondition loop

Start at	1	1	2	2
End at	4	5	4	5
Jump to	2	…	2	…

As demonstrated by the charts above, a precondition loop may have two jumps whereas a postcondition loop may have only one. Additionally, a compiler may unroll a loop into a repetitive sequence of expressions if certain compiler-specific conditions are met. Therefore, in general it is impossible to predict how many LCSAJ sequences are needed to cover a loop too.

To sum up, for some languages such as Haskell, which uses a completely different execution model and is also translated into native object code, it might be more difficult to define translation rules, whereas for other languages such as Java, which is translated into Java Intermediate Language, defining such translation rules might be feasible. In general, it is impractical to use LCSAJ for the analysis of source code written in a high-level language, although using this criterion for coverage analysis at object code level may be beneficial because it is more comprehensive than both decision and statement coverage but avoids exponential increase of cases as it happens with path coverage (Cornett 2011).

3.2.8 Path coverage

Path coverage measures whether each possible path in a method body was followed (Cornett 2011). It is a strong criterion which has two disadvantages. Firstly, the number of paths grows exponentially with the number of decision points, leading to explosive increase of the number of required test runs. Secondly, in practice many paths may be not feasible (Woodward 1980). For example, in the following code fragment, path coverage considers four paths, whereas only two of them might be feasible.

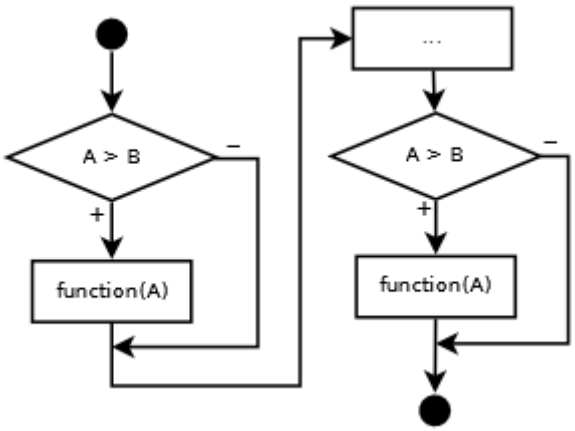


Figure 7. Path coverage example

3.2.9 Synchronization Coverage

Synchronization coverage is a less well-known criterion, which was first proposed by researchers from the IBM Haifa Research Lab (Bron – Farchi – Magid – Nir – Ur 2005) in an attempt to evaluate how well tests exercise concurrently executed program parts. Synchronization coverage measures if at each contention point, such as lock acquisition, waiting on a semaphore or in a monitor, there were both non-blocked and blocked threads. The idea behind this metric is to ensure that source code has been verified in concurrent environment. Furthermore, according to Microsoft Parallel Computing Platform engineers in practice many tests intended to verify concurrency aspects fail to do so reliably (Dern – Tan 2009).

Since synchronization coverage has four possible states (uncovered, unblocked, blocked, both) similar to decision coverage, it can be viewed as its special kind and said to measure decision coverage of certain internal conditional expressions inside concurrency library, which itself is not being tested. The simple programming language described in the beginning of this chapter does not specify any concurrency mechanisms, but if it did synchronization coverage would be overall similar to decision coverage. In other existing programming languages, concurrency mechanisms might be specified differently and therefore precise meaning of synchronization coverage might need to be clarified. Finally, it should be noted that synchronization coverage does not address other common concurrency issues such deadlocks and livelocks, just as decision coverage does not say anything about the presence of infeasible paths in the program under test.

3.3 Comparison of coverage criteria

Some of the described criteria are stronger than others and are said to imply or subsume them, whereas others have common cases but are not strictly ordered. Clarke, Podgurski, Richardson and Zeil described subsumption relationships between various data-flow coverage criteria in 1989, whereas the work of Woodward and Hennel from 2006 provides an overview of subsumption relationships between several control-flow coverage criteria. However, a slight change in definition of a coverage criterion may affect its subsumption relationships. For instance, decision coverage implies statement coverage in case no unconditional jumps, such as `goto` statements or `try-catch` blocks are used. On the other hand, statement coverage implies decision coverage in case all `if-then` statements have an `else` branch, and there are no empty branches. In general case decision and statement coverage are intersecting, but are not strictly ordered. The diagram below depicts general ordering of control-flow code coverage criteria according to the definitions given above.

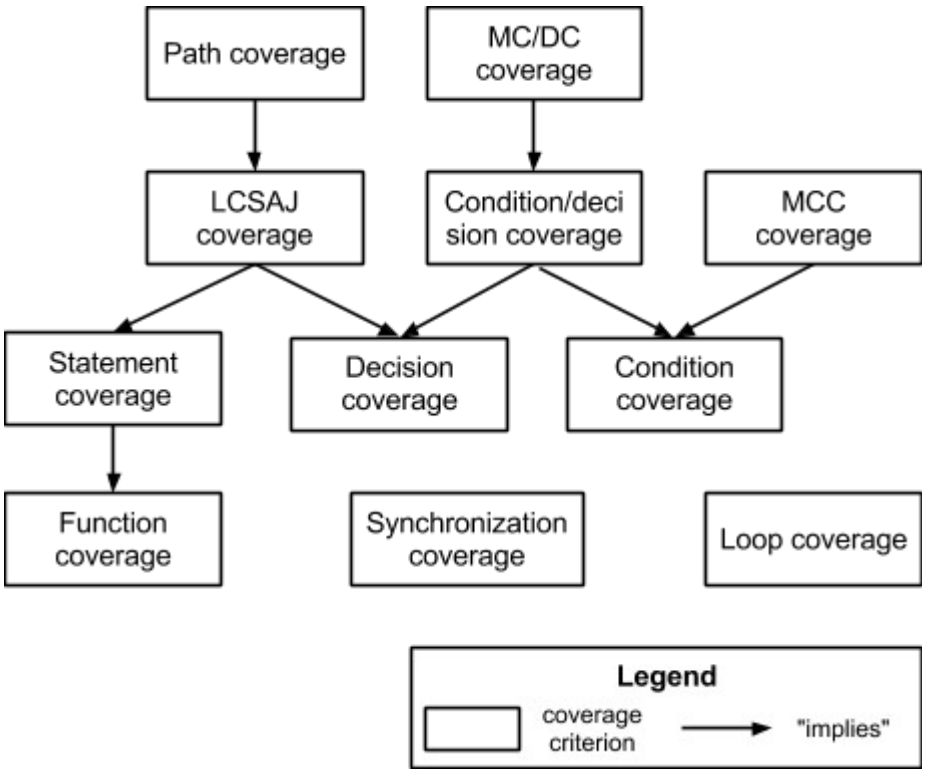


Figure 8. Relationships between coverage criteria

3.4 Applicability of coverage criteria

Software has to be tested and the efficiency of testing has to be verified regardless of the language in which it was written. Therefore, one may wonder if the coverage criteria described above can be equally well applied to programs written in mainstream imperative languages such as C or Java (King 2011). It is also unclear if they can be used with less popular languages such as Haskell.

3.4.1 Imperative languages

Java, C, C++, Perl and Python are typical imperative languages, even though some of them may include features more common among other kinds of languages. In general the above mentioned criteria are applicable across these languages, but their definitions may need be extended to accommodate language specifics. Furthermore, coverage tool authors sometimes alter these definitions slightly in order to improve them. The following cases demonstrate these concerns.

According to the function coverage definition above, a function is executed if it is entered at least once. The gcov tool for C report both the number of times each function was entered and the number of times it has returned (Free Software Foundation 2010a) so that the tester can decide him- or herself if function was sufficiently exercised by tests, whereas The BullseyeCoverage tool only reports if each function or lambda-expression was entered at least once (Bullseye Testing Technology 2013). Therefore, these two tools support different definitions of function coverage, and it is unclear if lambda-expressions are considered functions by gcov as they are by BullseyeCoverage.

Another question is how the coverage of a C++ template function is measured given that it is compiled into several instances. The authors of Bullseye Coverage tool have decided that the coverage of all instances should be combined and reported together (Bullseye Testing Technology 2013). The same logic can be applied to functions defined in closures, which are supported by Python.

When one looks at the loop coverage, he or she may wonder if a loop was executed once, and this depends on whether the loop has a condition in the beginning, in the middle, or in the end. It is unclear how tools, which support loop coverage, address this issue, but a reasonable solution could be to say that it was executed once if the main part of the loop was executed completely at least once.

Statement, line and block coverage also depend a lot on the programming language and tool used. For example, gcov reports line and basic block coverage where a line is either covered or uncovered, and a basic block is a sequence of statements, which does not contain decisions or function calls (Free Software Foundation 2010a). On the other hand, JaCoCo reports line coverage of java programs, where each line can be either uncovered, partially covered or fully covered, and the state depends on whether all java byte code instructions corresponding to a given line, were executed (Mountainminds GmbH & Co. KG and Contributors 2013).

Condition/decision coverage criteria have in practice especially many extensions. One question to resolve is if conditions outside decision blocks should be considered for coverage. If the following C program is exercised twice, firstly with x=1, and secondly with x=4, CTC++ coverage tool reports 100% coverage, whereas BullseyeCoverage reports 83%, because CTC++ does not consider conditions x>2 and x<5, whereas BullseyeCoverage does (Holzer – Schallhart – Tautschnig – Veith 2009, 2).

```
1  void foo(int x) {
2      int a = x > 2 && x < 5;
3      if (a) { 0; } else { 1; }
4  }
```

Listing 2. Code sample demonstrating the difference in C/D coverage interpretation

Additionally, for C++ try-catch-finally blocks BullseyeCoverage reports whether the try block is executed completely, and if each exception handler was entered. This is in contrast to JaCoCo java coverage tool, which does not consider try-catch blocks to be branches (Mountainminds GmbH & Co. KG and Contributors 2013).

Both MC/DC and LCSAJ coverage are measured on object code. According to DO-178 certification requirements it is permissible to measure MC/DC on source code only if object code can be clearly

traced at the source code level (Hayhurst – Veerhusen – Chilenski – Rierson 2001, 11). As for LCSAJ, it is difficult to apply this criterion to source code as was documented above.

Path coverage and synchronization coverage are not as widespread as the above named criteria and therefore the authors of this paper do not know how they are implemented in practice. However, it is conceivable that depending on the concurrency library used by software under test, tool vendors would have to redefine synchronization coverage to better suit the environment.

To sum up, most coverage criteria are defined in language-neutral way and need clarification before they can be applied to a given programming language. At the same time, some coverage criteria are defined with the assumption of a specific language or environment and cannot be applied if these assumptions do not hold. So, language specifics make it difficult to describe coverage criteria unambiguously in a language-neutral way, whereas coverage tool vendors disagree about adaptation of criteria to a single language.

3.4.2 Functional languages

Support for coverage analysis is also available for software written in functional languages such as Haskell and Erlang. To start with, the popular Glasgow Haskell Compiler (GHC) includes Haskell Program Coverage (HPC) tool since version 6.8.1 (The GHC Team 2007). The HPC tool is able to report the coverage of all declarations, alternatives, expressions and Boolean contexts. Since functions are first-class objects, and variables are assigned only once in Haskell, the "declarations coverage" covers both of them and roughly corresponds to function and a sub-part of statement coverage in a traditional imperative language. "Alternatives and Boolean context coverage" are HPC counterparts to decision and condition coverage discussed above. Finally, expressions coverage by most replaces statement coverage since Haskell programs are composed of expressions rather than of statements. It is interesting to note that HPC authors considered Boolean data type to be no special case, and contemplated adding more coverage criteria for other data types (Gill – Runciman 2007, 10). Boolean requires less special handling in Haskell compared to traditional imperative languages because control flow in Haskell is determined by pattern matching and lazy evaluation rather than by explicit branching and consecutive computation based on side-effects.

Erlang Open Telecom Platform includes 'cover' module (Ericsson AB 2012), which is probably the only existing publicly available coverage tool for Erlang (Widera 2005, 152). It supports function and line coverage, which are defined according to the description above. Line coverage is more applicable for programs written in Erlang than in Haskell because of eager evaluation which makes it easier to understand the control flow by looking at the source code. Therefore, it is possible to reason why a given line of source code was not covered by looking at nearby lines, which is more difficult in Haskell. Nevertheless, line coverage does not provide a good level of assurance of test efficiency, and it was proposed to augment it with data flow coverage for Erlang programs (ibid.).

In general, coverage analysis is as important for functional programs as it is for imperative ones. However, coverage criteria used for imperative programs may be less relevant for functional ones due to differences in control flow (lazy vs. eager evaluation) and prevailing branching methods (Boolean if-statements vs. pattern matching).

3.5 Related work

Andreas Holzer, Christian Schallhart, Michael Tautschnig and Helmut Veith from TU Darmstadt have created a generic coverage criteria specification language FQL and its implementation FShell (Holzer – Schallhart – Tautschnig – Veith 2009). Their tool parses an ANSI C program, produces its control flow graph, and finds test inputs which trigger execution of certain paths in the program and thus provide required coverage level. It may become a useful tool for minimizing test suites and improving their completeness, however, it does not yet allow specification of properties required to express such coverage criteria as MC/DC (ibid., 28), and has "limited support for function calls by function pointers and no support for longjmp and setjmp" (ibid., 26), nor has it been tried with other languages. On one hand, extending the FQL language with path set predicates required for MC/DC specification is feasible. On the other hand, extending FShell to support dynamic control flow automata necessary for the dynamic features of C language may require further research. Finally, producing a control flow automaton for a program written in a functional language such as Haskell is complicated (Midtgaard – Jensen 2012), whereas for such language as Perl 5 it is impossible without executing the program (Kegler 2008). Indeed, the authors of the paper themselves state that the "behavior left undefined by the C standard is fixed in an arbitrary manner" (Holzer et al. 2009, 26), which points to the fact that many existing programming languages may be not amenable to precise language-neutral specification of coverage criteria.

Tim Miller, Padaghm Lin, John Thangarajah, Danny Weyns and Marie-Pierre Gleizes have also compared coverage criteria to each other, and although they used different notation, the end results are comparable to ours (Miller – Padgham – Thangarajah – Weyns – Gleizes 2011). Martin R. Woodward and Michael A. Hennell have further developed the concept of LCSAJ coverage to so-called JJ-path coverage, and compared it with MC/DC coverage, also concluding that they are in general incomparable, although under certain conditions JJ-path coverage subsumes MC/DC (Woodward – Hennell 2006).

3.6 Conclusions and further work

Coverage analysis is a widespread method to assess test efficiency, and there are coverage tools for various programming languages. Many of them provide similarly named coverage criteria, which are in practice quite different. Some of these differences are caused by language specifics, and some are introduced by tool authors on their own initiative. It is clear, however, that there are no common coverage criteria definitions and even though there exist unification proposals, they have not received wide adoption or been implemented yet. Another important theoretical question is which code coverage criteria are feasible and useful for programs written in non-imperative languages such as Haskell, Erlang, Scheme, Smalltalk or Prolog. Finally, in order to be able to reason about usefulness of various code coverage criteria, one has to understand use cases for code coverage data, and this subject is discussed in the next chapter.

4. Relationship between code coverage and test suite qualities

Code coverage measurement captures facts about the test suite and the software under test. These facts are commonly used to assess test suite completeness, i. e. if it includes test cases to cover all parts of software under test (Zhu – Hall – May 1997). In this chapter several other test suite qualities are defined and the connection between them and code coverage is examined.

4.1 Test suite qualities

Software verification is an indispensable part of software development. One of the artifacts produced during verification is a set of test cases. A test case is a formal detailed sequence of steps to validate correct behavior of developed software. A good test case should have certain qualities, and be

- necessary,
- correct,
- falsifiable,
- traceable to its source of origin,
- independent from other test cases,
- atomic,
- efficient,
- maintainable, and
- consistent in the value it returns.

Individual test cases are grouped together into test suites based on testing method, tested functionality, level or scope of testing. Consequently, large software products may have several test suites composed of unit, functional, integration and acceptance tests. A good test suite should also possess certain qualities such as being

- complete, i.e. testing all the necessary features,
- effective, i.e. being able to find defects,
- minimal, i.e. not redundant,
- consistent, i.e. having no test cases with conflicting requirements,
- modifiable, i.e. consisting of atomic, independent, maintainable test cases,
- regression-sensitive, i.e. able to quickly detect new defects introduced into previously tested code.

Relationship between code coverage and test suite qualities has been the focus of a number of research efforts, which proved that code coverage analysis can be efficiently used to estimate percentage of uncovered defects, aid in prioritization of test cases during regression testing, support elimination of unnecessary test cases and even participate in the creation of test cases.

4.2 Defect detection

The most important characteristic of a test suite is its capability to detect defects. Originally, code coverage was introduced as a method to evaluate the completeness of a test suite with the assumption that a complete test suite indicates the lack of defects in the tested software.

Phyllis G. Frankl and Oleg Iakounenko have measured test effectiveness of groups of randomly generated test suites which provide the same coverage level. They defined test effectiveness as a proportion of test suites in a group, which detect at least one error, and found a strong correlation between the group coverage level and its test effectiveness. Furthermore, they observed variation of correlation coefficients among different subject programs but noted that "all eight of our subjects showed both all-uses and branch coverage to be significantly more effective than random testing". (Frankl – Iakounenko 1998)

Saiyoot Nakkhongkham shows significant positive association between the achieved level of code coverage and the number of found pre-release defects as well as negative correlation between the level of code coverage and the number of defects found after the release, which implies positive software and test suite quality (Nakkhongkham 2011).

The fact that high code coverage is necessary but not sufficient for well-tested code is known in the industry (Marick 1997). Assuming that tests were written from requirements and are able to detect errors of omission, one reason why high code coverage still does not guarantee that code is well tested, is because it does not take into account quality of test cases themselves. However, further assuming that tests are of high-quality, a question about the quantitative relationship between the number of undetected defects and the level of code coverage remains.

One might guess that the number of detected defects grows linearly with the code coverage percentage, but results obtained by Frankl and Iakounenko demonstrate that the probability of defect detection starts to grow only when a certain minimal level of coverage is obtained (Frankl – Iakounenko 1998). Williams et al. started from the observation that code quality often grows exponentially with code coverage, and tried to devise a model explaining this phenomenon. In order to simplify derivation, researchers assumed that all tests are equal and always detect defects in the statements they test, and that defects are distributed uniformly and do not depend on each other. It turned out that under such assumptions, the probability of a tested program being free from undetected defects is an exponential function of code coverage, and is given by the following equation (Williams – Mercer – Mucha – Kapur 2001):

$$Q = (1 - P_n)^{n(1-T)}$$, where

Q is probability that a tested program does not have any undetected defects;
P is probability that any single statement has a defect;
n is the total number of statements in the tested program;
T is the code coverage percentage.

The following figure shows the general relationship between the level of code coverage and code quality.

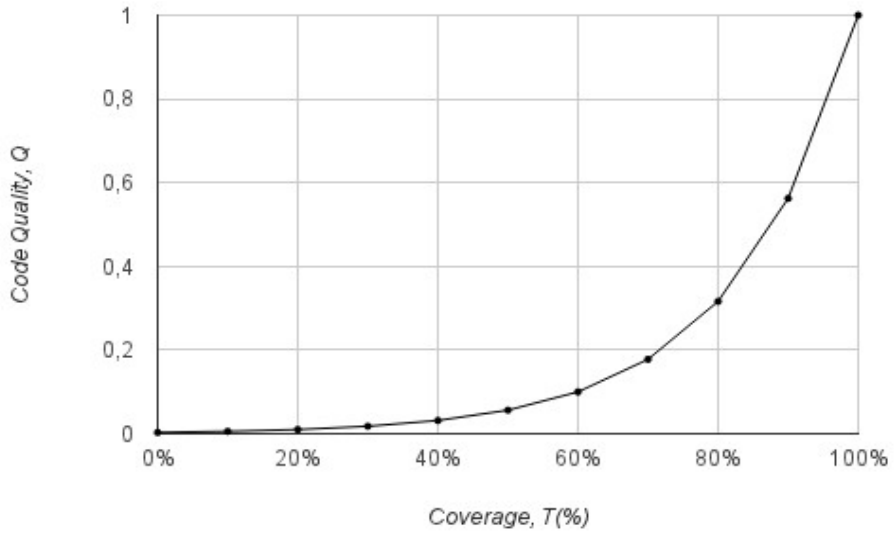


Figure 9. Path coverage example

In other words, the probability that a program is bug-free with 50% coverage is 4.9% if one expects to have 6 defects per 1000 statements, and the program consists of 1000 statements. In order to achieve a 50% probability of a bug-free program, he or she has to reach 88.5% coverage. It is noteworthy that this result is applicable not only to statement coverage, and it is suggested that any units of code can be employed as long as they are the ones, which were used to measure code coverage.

However, in practice tests are not perfect, and defects are not uniformly or independently distributed throughout the code either. Joel Troster of IBM Canada has studied relationship between various code metrics and defect history of commercial software comprised of over 500000 source code lines developed in the course of 22 years, and observed statistics for the number of defects per one thousand lines of code ("Defects per KLOC") provided in Table 12 below. It follows from this data that more than half of code chunks were free from defects, whereas few chunks had a large number of them. Furthermore, according to Chebyshev's inequality ca. 89% of chunks should have less than 46.4 defects. (Troster 1992)

Table 12. Defects per KLOC statistics from Troster's study

Mean	Standard deviation	Median	Minimum	Maximum
5.6	13.6	0	0	250.0

Therefore, Troster's empirical data support the hypothesis that defects are more likely to cluster around some places rather than others. He goes on to analyze correlation between the number of defects and various code metrics, and concludes that the single most efficient metric is the number of changed source instructions, which has a Spearman correlation coefficient of 0.71 with the number of defects per thousand source code lines. (ibid.)

Therefore, although code coverage is a good indicator of a test suite completeness, it is important to take into account defect clusterization.

4.3 Regression sensitivity

It turns out that code coverage can be used to prioritize test cases in a suite in order to minimize time it takes for the suite to detect a recently introduced regression. One example of commercial software providing this feature is Clover. It provides a testing facility, and tracks which lines of code were changed since the last test run with coverage. The tests that cover changed lines are run first so that any trivial regressions are quickly detected and reported to the developer.

Gregg Rothermel et al. studied prioritization of test cases, which does not require integration with developer's testing facility. They compared the number of defects detected by a test suite in a given amount of time, and concluded that "natural" order of test cases yields the worst regression sensitivity. Randomly sorted test cases provide better regressions detection time than natural order, and sorting test cases in the order of decreasing coverage resulted in the shortest times. It is noteworthy that neither the choice of coverage criteria (statement or branch) nor the choice of sorting criteria (by total coverage or by unique coverage) created a noticeable impact on the resulting time. (Rothermel – Untch – Chu – Harrold 2001)

4.4 Test suite minimization

Minimality of a test suite is its another important quality, which contributes to its maintainability and shorter running times. However, it is not a trivial task to decide which tests are testing already tested code, and if they are excessive in such a case. It is conceivable that on some occasions several tests might be required to test a single statement, or that spending time minimizing test suites could be a cost-ineffective task. Nevertheless, it is unclear what happens if one eliminates test cases, which do not add any unique coverage to the suite.

W. Wong et al, have researched this question and discovered that when test suite is reduced while code coverage is kept constant there is little or no reduction in the defect detection. It means that code coverage measurements can be used for test suite minimization with little loss in fault detection capability. (Wong – Horgan – London – Mathur 1995)

4.5 Test case generation

Finally, code coverage can be used to guide generation of missing test cases. Holzer et al. describe one such approach, which involves providing specification of coverage requirements in FQL to a bounded model checker, which in turn produces a set of test cases that satisfy requirements (Holzer – Schallhart – Tautschnig – Veith 2009). Tavis Ormandy has reported successful use of assembly instruction level code coverage to guide fuzz testing at Google (Ormandy 2009).

4.6 Summary

It has been demonstrated that code coverage data can be used

a). to assess test suite completeness and effectiveness;
b). to improve regression sensitivity of a test suite;

c). to minimize a test suite in order to decrease its running time or to improve maintainability;
d). to guide test case generation in order to quicken development or to perform fuzz testing.

At the same time, there is not enough research to support or refute efficiency of using code coverage data to improve test suite modifiability, although the fact that it can be used to minimize a test suite is promising.

It is also worth noting that the relationship between attained code coverage level and probability of a program being bug-free is not linear but exponential. However, since defects are distributed non-uniformly, it is also important to weigh in defect and code change history when assessing the probability of a piece of software containing a defect.

5. Presentation of coverage data

A case study of several popular code coverage tools is performed in this chapter. Its first goal is to investigate which coverage criteria they support, and its second goal is to analyze which test suite qualities can be measured or affected using the code coverage data from these tools. In order to make this analysis concrete, the following user stories are utilized.

1. **Test suite completeness:** As a developer, I want to know how much different parts of software have been exercised by tests so that insufficiently tested components are revealed.
2. **Test suite minimization:** As a developer, I want to run only relevant test cases so that running tests takes less time.
3. **Test suite prioritization:** As a developer, I want the first few test cases to detect most regressions so that I get a quick feedback.
4. **Test suite maintainability:** As a developer, I want to know which test cases cover which parts of software under test so that I can spot redundant test cases as well as quickly see if a place where a defect was found was already covered by a test case.
5. **Test case generation:** As a developer, I want test cases to be generated automatically so that I only need to verify them and don't spend time on writing.

The tools selected for this case study are used to analyze coverage of software written in such widespread programming languages as C/C++ (BullseyeCoverage and CTC++) and Java (EclEmma and Clover). It was assumed that the most popular languages should have the best tools and popular tools with long development history would include advanced features.

5.1 EclEmma

EclEmma is an open-source plug-in for Eclipse IDE that was originally based on EMMA code coverage tool, which has been replaced with JaCoCo code coverage library in recent releases (Mountainminds GmbH & Co. KG and Contributors 2011).

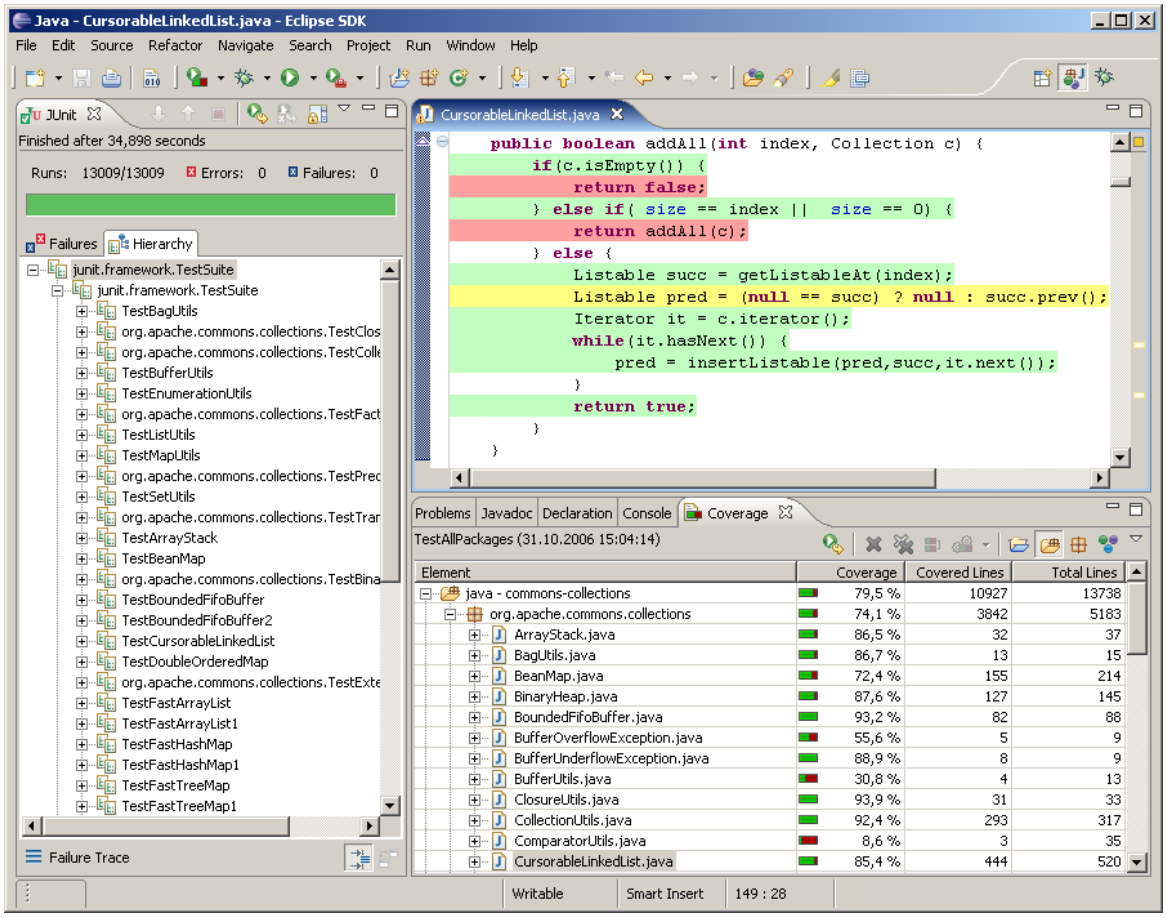


Figure 10. EclEmma screenshot
Copyright: Mountainminds GmbH & Co. KG and Contributors 2011

EclEmma measures coverage at Java byte code level but annotates source code line by line. Lines corresponding to completely covered and completely uncovered byte code, are colored green and red respectively. Those lines, which correspond to partially exercised byte code are colored yellow (Roubtsov 2006). There is no way to say which test cases have covered a particular line, and no way to say which parts of a yellow line were executed and which were not.

EclEmma coverage statistics view organizes Java source code in a tree and augments each node in the tree with information about its aggregate coverage measured in several units: percentage of covered lines, number of covered lines, number of uncovered lines and total number of lines. Furthermore, it is possible to select which coverage criteria should be utilized from byte code instructions, branches, lines, methods, types, and cyclomatic complexity. Cyclomatic complexity coverage refers to branch coverage aggregated using a special formula (Mountainminds GmbH & Co. KG and Contributors 2013).

This tool addresses only the test suite completeness user story through its source code annotation and a coverage statistics view.

5.2 Clover

Clover is a commercial code coverage tool for Java and Groovy developed by Atlassian (Atlassian 2013). It measures coverage using statement, decision and function criteria, and lets its user decide which metrics he or she is interested in. Like many other tools Clover employs source code annotations and an aggregate coverage statistics view to visualizes its measurements. Similar to EclEmma it has the concept of partial coverage, but unlike EclEmma, Clover has a notion of Working Sets, which make it straightforward to specify boundaries of software under test.

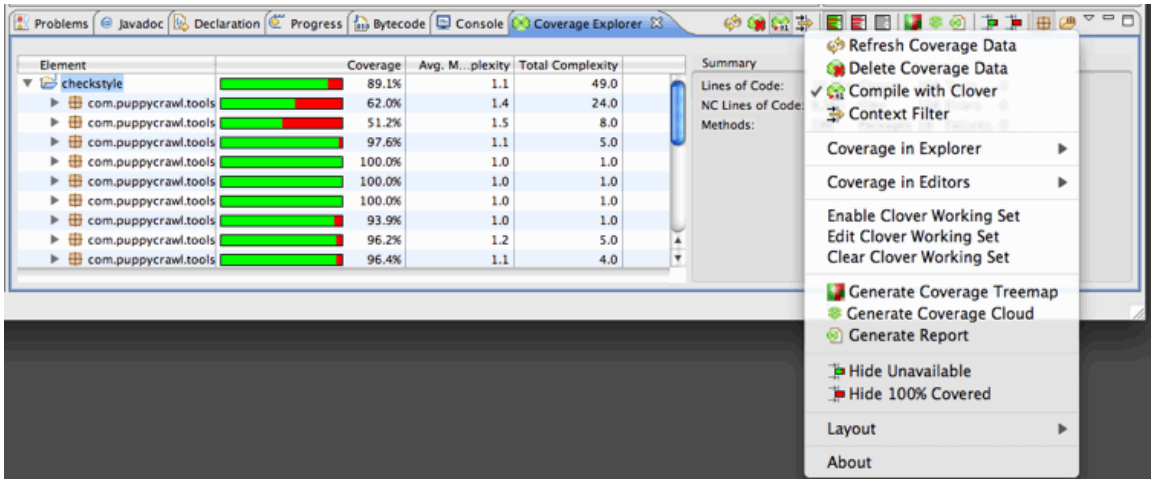


Figure 11. Clover Coverage Explorer view and Working Set menu options
Copyright: Atlassian 2013

When it comes to running tests, Clover can automatically select relevant test cases to be run based on test failure history, code change history, or provided coverage level, which addresses the test suite minimization user story. Clover is also able to reorder test cases in order to improve regression sensitivity or ensure absence of interrelationships, which satisfies the test suite prioritization user story. Both of these options are available via so-called optimized launch configurations.

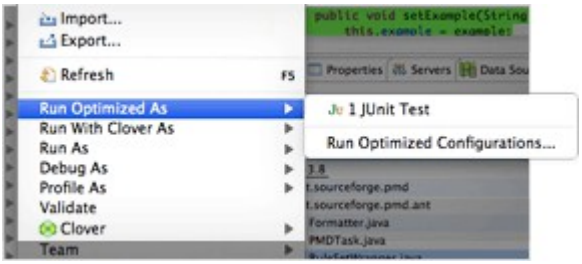


Figure 12. "Run optimized as" menu
Copyright: Atlassian 2013

Furthermore, Clover records coverage provided by each test case and makes it available to programmer via source code annotation and a pop-up window. Therefore, it also satisfies the test suite maintainability user story.

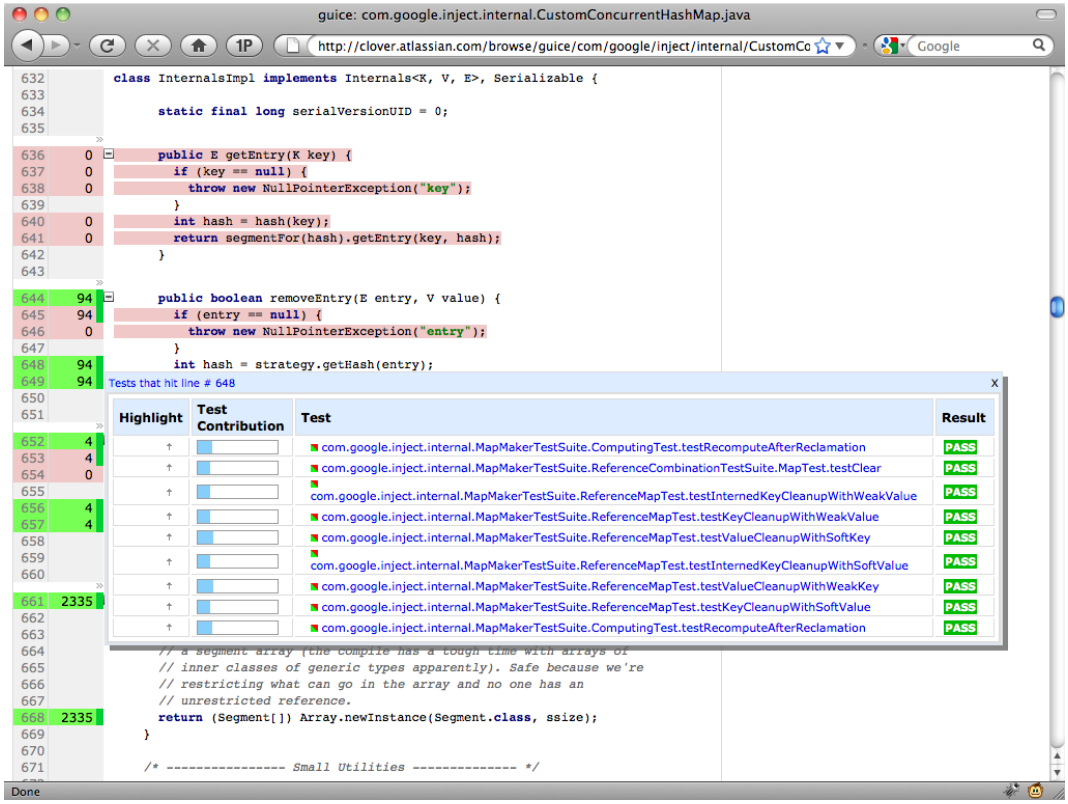


Figure 13. Source code annotation and the list of tests, which exercised a given line
Copyright: Atlassian 2013

In addition to the aforementioned features, Clover measures code complexity and makes it easy to find Java classes with high cyclomatic complexity and low coverage level using Coverage Tree Map and Class Complexity diagrams.

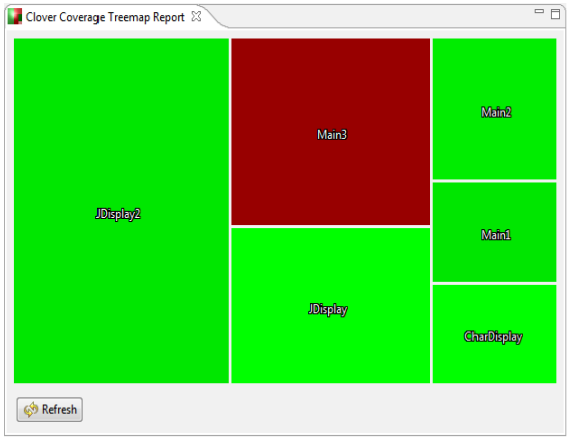


Figure 14. Clover Coverage Treemap report

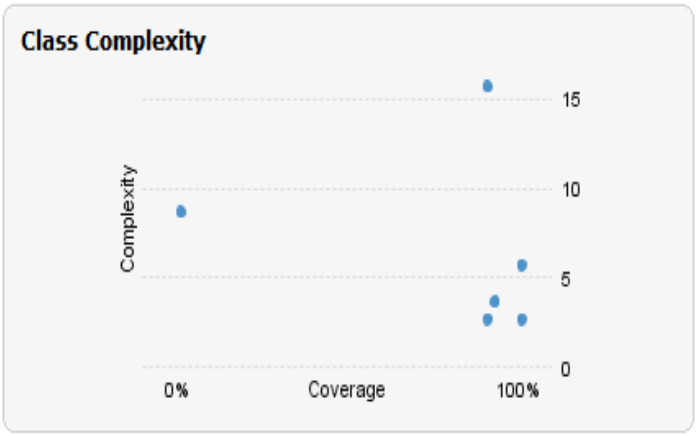


Figure 15. Clover Class Complexity report

Clover Coverage Treemap is a hierarchical network map (Mansmann – Vinnik 2006), which shows comparative complexities of different packages and classes along with their coverage metrics. The size of each rectangle is proportional to the cyclomatic complexity of the class and the color depends on coverage percentage. Clover Class Complexity view is a two-dimensional scatter plot with X-axis representing coverage and Y-axis representing cyclomatic complexity.

Finally, Class Coverage Distribution and Coverage Cloud views along with Trend Reporting tool and the Clover Dashboard simplify getting an overview of test suite completeness status for the whole project.

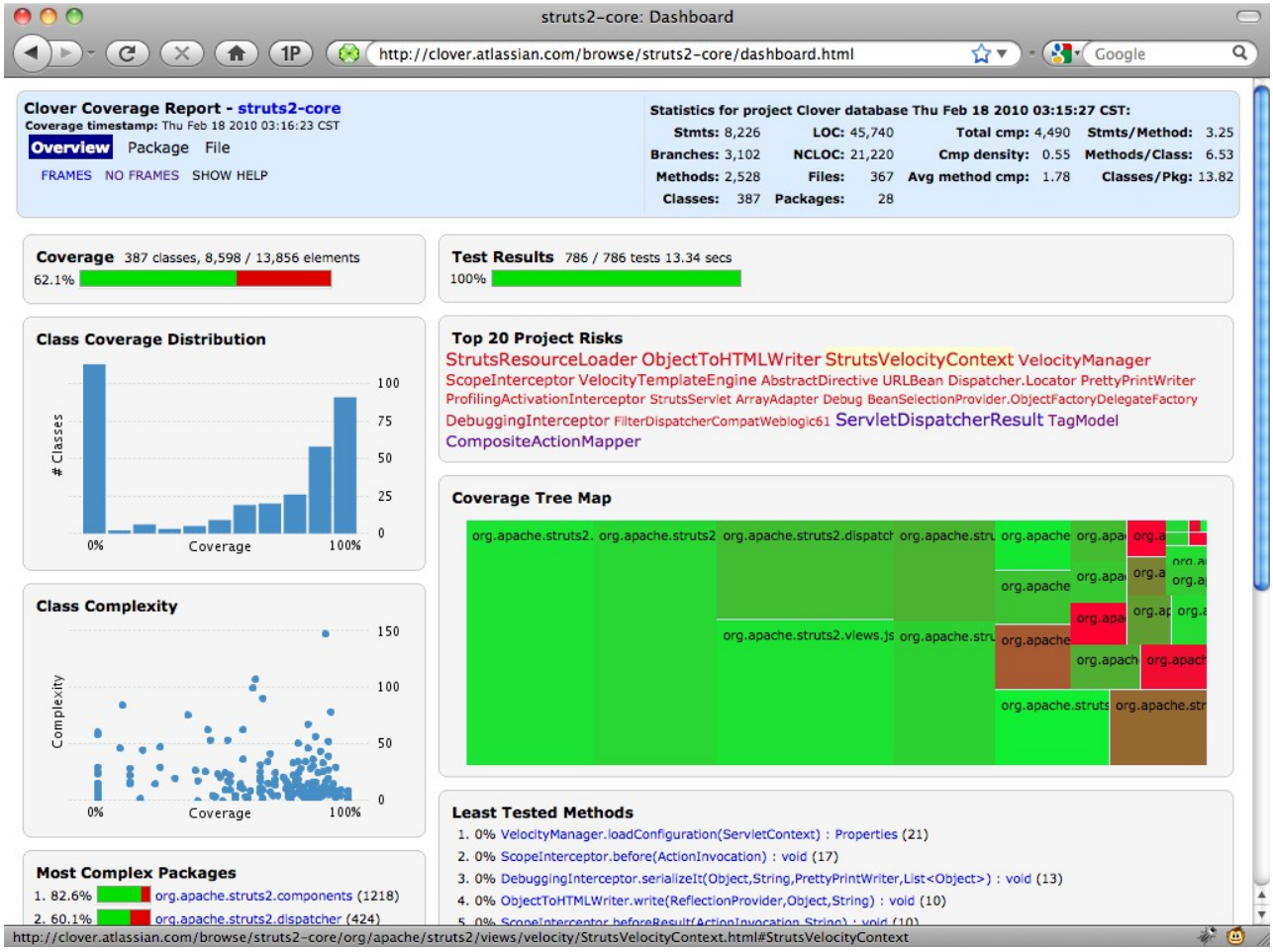


Figure 16. Clover Dashboard
Copyright: Atlassian 2013

Despite satisfying the test suite minimization, prioritization and maintainability user stories, Clover cannot automatically classify tests based on whether they cover a single unit, a group of units, or a complete application. Therefore, it cannot be used for example to automatically generate test suites optimized for running at different levels of continuous integration process. It is also noteworthy that Clover does not support automated test case generation.

To sum up, Clover addresses test suite completeness, minimization, prioritization and maintainability user stories but lacks support for automated test case generation.

5.3 BullseyeCoverage

BullseyeCoverage is a commercial tool from Bullseye Testing Technology for measuring and analyzing code coverage of software written in C and C++. In contrast to EclEmma, which

measures coverage according to function and statement coverage criteria, BullseyeCoverage measures coverageе using function and condition/decision coverage metrics.

BullseyeCoverage instruments source code after it was parsed by a preprocessor but before it reaches a compiler, and only measures entry points. (Bullseye Testing Technology 2013) Therefore, BullseyeCoverage is not affected by the partial coverage problem of EclEmma.

BullseyeCoverage can output coverage analysis results to standard output as well as show them in a Coverage Browser viewer software. The latter offers annotated source code as well as aggregated coverage statistics views. However, instead of highlighting source code, a column with coverage information is attached to read-only plain text code viewer, as shown below. Please, note how line 67 in the code viewer is split into several ones.

```
✓    52 int main()
     53 {
     54     int c;
     55     int temp;
     56     bool quit = false;
     57     printf("Enter reverse-polish expressions, q to quit\n");
     58     Stack stack;
TF   59a    while (
  tf 59b          !quit &&
 →t  60       (c = fgetc(stdin)) != EOF)
     61    {
     62        switch (c) {
✓    63        case ' ':
✓    64        case '\n':
     65            break;
     66
→    67a       case '0':
✓    67b                 case '1':
✓    67c                           case '2':
✓    67d                                     case '3':
→    67e                                               case '4':
```

Figure 17. Source code annotation for C/D coverage in BullseyeCoverage Browser

The coverage statistics view below is similar to the one used by EclEmma and Clover.

Name	Function coverage	Uncovered functions	Condition/decision coverage	Uncovered conditions/decisions
eval(char*,unsig...	100%	0	50%	3
lex()	100%	0	19%	29
main()	100%	0	60%	4
match(int)	0%	1	0%	4
name(int)	0%	1	0%	11
prs(unsigned)	100%	0	60%	53
z(unsigned long)	100%	0	25%	3

Figure 18. Coverage summary for C/D coverage in BullseyeCoverage Browser

The viewer lets its user filter coverage statistics using several parameters, and accompanying command-line tools let one automate this process in order to only show coverage for software under

test. Unfortunately, because of wide diversity among C and C++ testing frameworks BullseyeCoverage is not integrated with any one, and thus lacks data needed to meet test suite minimization, prioritization and maintainability user story. Automated test generation cannot be implemented in BullseyeCoverage for the same reason.

To sum up, BullseyeCoverage addresses only the test suite completeness user story and lacks support for other ones.

5.4 CTC++

TestWell CTC++ is another commercial code coverage tool for C and C++ software, which supports condition/decision (C/D) coverage as well as more advanced multicondition coverage (MCC) and modified condition/decision coverage (MC/DC) criteria. (Testwell 2013a) It works similarly to BullseyeCoverage by instrumenting source code before it is passed to a compiler. However, CTC++ presents coverage results in a different way.

Before instrumentation a user has to specify what kind of coverage data CTC++ should collect: function, decision or multicondition coverage. After software has been tested CTC++ saves coverage measurements into a file. This file can be converted into various formats including plain text and HTML (Testwell 2013a). CTC++ does not have a special viewer for these files and relies on text editor and HTML browser installed on developer's machine. CTC++ integrates several controls into Visual Studio but they also rely on HTML browser to present coverage information. Therefore, HTML report provides richest interface and is examined below (Testwell 2013b).

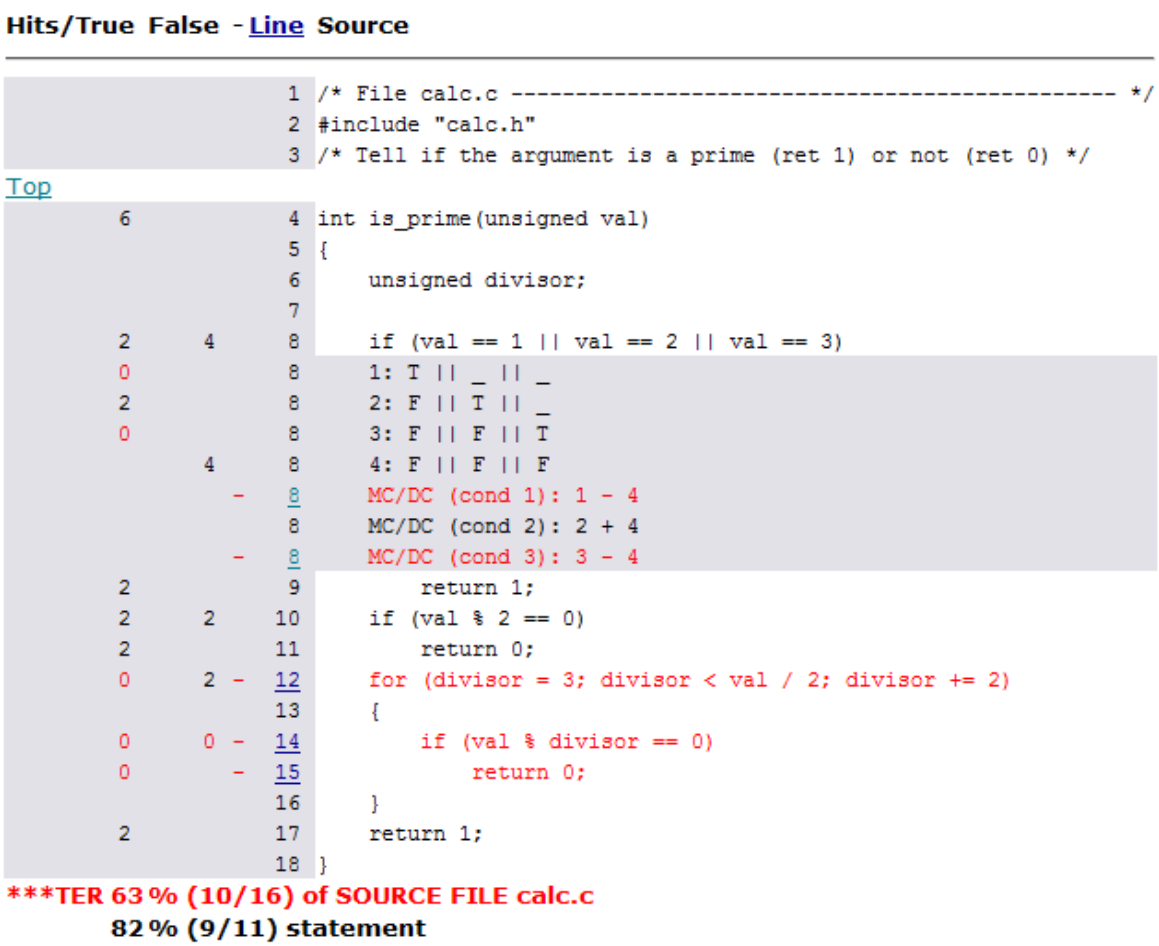
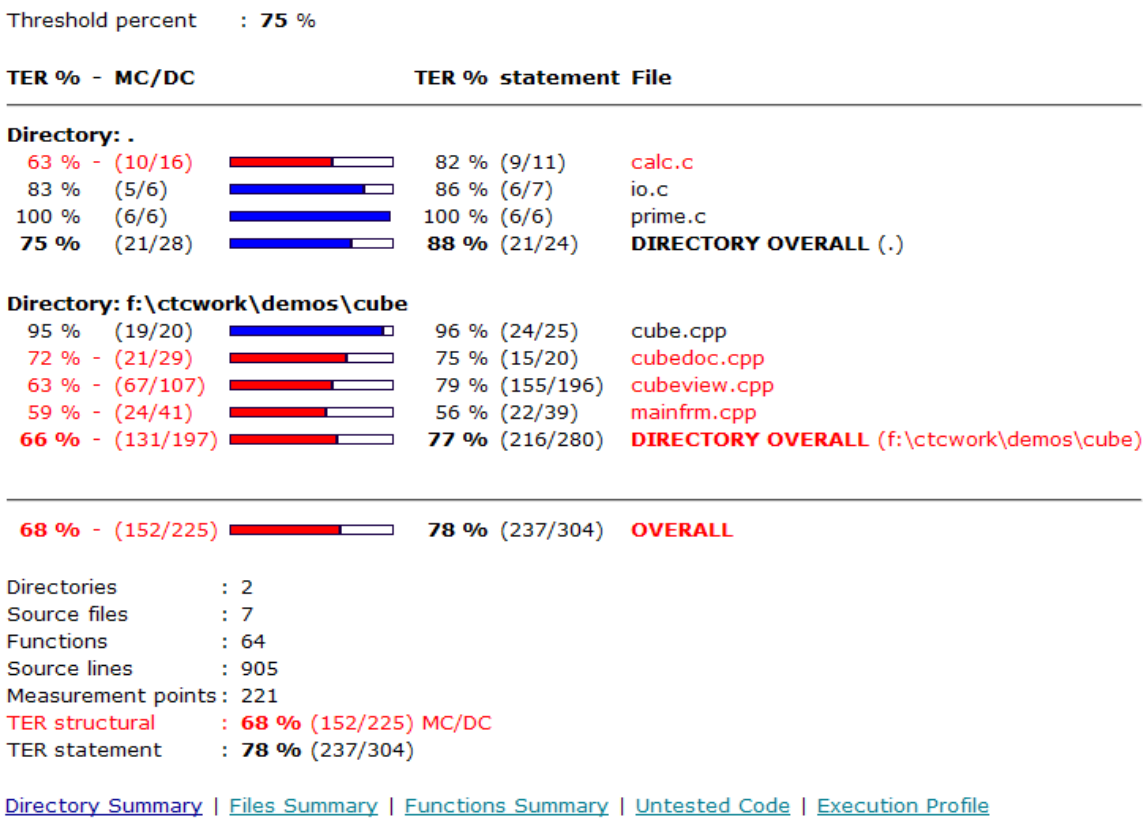


Figure 19. Source code annotation for MC/DC coverage in CTC++

CTC++ uses a combination of source highlighting with text insertion to visualize code coverage data. Furthermore, the results are aggregated across functions, files and directories as shown on the next screenshot.



Threshold percent : **75** %

TER % - MC/DC		TER % statement	File
Directory: .			
63 % - (10/16)		82 % (9/11)	calc.c
83 % (5/6)		86 % (6/7)	io.c
100 % (6/6)		100 % (6/6)	prime.c
75 % (21/28)		**88 %** (21/24)	**DIRECTORY OVERALL** (.)
Directory: f:\ctcwork\demos\cube			
95 % (19/20)		96 % (24/25)	cube.cpp
72 % - (21/29)		75 % (15/20)	cubedoc.cpp
63 % - (67/107)		79 % (155/196)	cubeview.cpp
59 % - (24/41)		56 % (22/39)	mainfrm.cpp
66 % - (131/197)		**77 %** (216/280)	**DIRECTORY OVERALL** (f:\ctcwork\demos\cube)
68 % - (152/225)		**78 %** (237/304)	**OVERALL**

Directories : 2
Source files : 7
Functions : 64
Source lines : 905
Measurement points : 221
TER structural : **68 %** (152/225) MC/DC
TER statement : **78 %** (237/304)

Directory Summary | Files Summary | Functions Summary | Untested Code | Execution Profile

Figure 20. Coverage summary for MC/DC coverage in CTC++

Similar to BullseyeCoverage, CTC++ satisfies only the test suite completeness user story despite supporting more coverage criteria. It neither assists programmer in minimizing and reordering test suite, nor simplifies test suite maintenance or provides test case generation. However, in contrast to BullseyeCoverage, Testwell CTC++ is only a part of a tool chain provided by the same company. This tool chain includes a unit testing framework for C++ called CTA++. So it should be possible to CTC++ with CTA++ in order to provide better functionality.

5.5 Conclusion

To start with, all four examined code coverage tools support test suite completeness analysis by annotating source code with coverage information and providing a coverage summary view. Only one of these tools goes a step further and also aids programmer in maintaining test suites as well as selecting and reordering test cases. None of the tools support automated generation of missing test cases.

It is noteworthy that in certain cases it may be difficult to utilize coverage data for anything beyond test suite completeness analysis because the programming platform lacks precise specification. For example, the lack of a standard unit testing framework makes it difficult to add these features to a C/C++ coverage analysis tool. Furthermore, implementation of a coverage analysis tool depends on build system and runtime interfaces. Therefore, it is desirable that a code coverage tool is developed together with a unit testing framework, or better yet planned for already at the language design stage.

Secondly, it is interesting that despite differences in precise definition of coverage criteria most tools present coverage information in a similar way. All of them annotate source code either using color-coding or by attaching a column with coverage information, and all of the tools provide a coverage summary view, which depicts aggregated coverage as a percentage of probes executed by tests to the total number of probes. The only tool, which supports test suite maintenance, shows information about related test cases either in a pop-up window or in a separate view.

Regardless of the way source code is annotated, a programmer benefits from integrating these annotations into his or her IDE for several reasons. Firstly, IDE simplifies understanding and navigating source code by providing such features as syntax highlighting and file outline. Secondly, IDE increases programmer productivity during code refactoring, thus making it easier to correct problems uncovered during code coverage analysis. Finally, fewer better integrated tools let programmer receive a quicker feedback for the code he or she is modifying. So, it is desirable for any coverage analysis tool to integrate into IDE.

Another observation from this case study is the uniform use of percentage numbers or bars as a measure of aggregated coverage. Based on the theoretical exponential relationship between coverage level and code quality (see Chapter 4.2 Defect detection), it is suggested to replace percentage with “coverage ranks”, which have logarithmic relationship to code quality and make coverage-quality relationship appear to be linear. For example, coverage rank 1 one can start at 50% coverage, coverage rank 2 can start at 75% coverage, coverage rank 3 at 88% etc. The scale doesn’t have to start at 50%, but it has to follow the same function.

Finally, many tools show coverage for all source code used to build executables or even for all source code which belongs to a project or a solution in an IDE. The boundaries of software under test may be different, in which case redundant files may distort aggregate coverage result and missing required files may stay insufficiently tested. Therefore, it should be possible to explicitly select which parts of software are under test so that coverage is measured and shown only for them.

6. Development of an BullseyeCoverage plug-in for Eclipse

Eclipse Platform is a modular framework for developing Java applications, which is well-known for Integrated Development Environments (IDEs) built upon it. Eclipse CDT is one such IDE for C and C++ development, which is extensively used by one of Purple Scout's customers. This customer also happens to use BullseyeCoverage tool to measure code coverage of his C and C++ code. However, coverage reports produced by BullseyeCoverage are printed to stdout, which makes them difficult to comprehend and discourages developers from performing coverage analysis. In order to overcome these difficulties, a Bullseye plug-in for Eclipse is requested by the customer.

6.1 Requirements

Initially, only two features have been requested from the plug-in.

- F01. Annotate source code with coverage information
- F02. Show aggregate coverage at class/file/directory level.

However, during discussion several other features have been identified.

- F03. Jump from an aggregate coverage item to the corresponding source file/class/function on a double-click.
- F04. Provide a way to sort aggregate coverage items
- F05. Provide a way to filter aggregate coverage items
- F06. Persist coverage information between Eclipse restarts.
- F07. Delete coverage information on user request.
- F08. Automatically delete coverage information when the report file is deleted.
- F09. Automatically reload coverage information when the report file is updated
- F10. Provide a way to globally hide and unhide coverage annotations.
- F11. Provide means to show merged coverage from several coverage reports.
- F12. Provide means to show differential coverage between several coverage reports.
- F13. Make color selection for merged and differential coverage a user preference.

Beside these features, the following non-functional requirements have been put forward

- N01. Parsing a report which describes 10000 source files á 100 lines should take less than 60 seconds, and the UI should not be blocked during this operation
- N02. Opening a source file with 10000 lines should take less than 5 seconds, and the UI should not be blocked longer than 1 second.
- N03. Any invalid input data should be handled and an error message should be printed.
- N04. The plug-in should not be loaded before it is actually required.

6.2 Development process and tools

Plug-in development was organized in one-week iterations, which started with a sprint planning each Friday, and a demonstration next Friday. Features have been divided into tasks in order to break work in chunks, which take one person less then a week. Sprint demonstrations were made to

and tasks were prioritized by the Purple Scout project manager. The first external demonstration for Purple Scout's customer representatives happened after six weeks.

Redmine was used for issue tracking, and Git was used for source code management. Eclipse Plug-in Development Environment (PDE) was used as an IDE, and Evolus Pencil was used for UI prototyping.

6.3 Domain analysis and design

6.3.1 Input data

BullseyeCoverage saves its measurements in a proprietary binary format, but comes with a converter, which can export the data into an XML report file. It turned out to be simple to make sure that this XML report file is produced when the coverage target is built.

In order to save time, and provide a robust solution, it was decided to use the Eclipse Modeling Framework (EMF), which has built-in support for XML de-/serialization for reading and writing report files. An EMF Ecore model has been created to represent data structure in a report file, and Java classes along with XML readers and parsers were generated for this model.

6.3.2 Target environment

The plug-in was supposed to be run with Java 6 in Eclipse Platform 3.5.2 + Eclipse CDT 6.0.0 environment. Since Eclipse is built on top of OSGi dynamic component framework, modularity is in its nature, and Eclipse Platform as well as Eclipse CDT are composed of independent OSGi bundles. Furthermore, both the platform and the CDT define numerous variation points, which let one contribute extensions for them by declaring extending classes in an XML file. These mechanisms make writing an Eclipse plug-in rather simple.

6.3.3 Design

After the problem of data import was resolved, the work proceeded along two parallel tracks. The first track focused on providing source code annotations, and the second track aimed to create a coverage statistics view.

It has been decided quite soon that source code annotations would be placed in a ruler to the left from the text area with source code instead of using colors to convey the same information. This should have simplified implementation of merged (F11) and differential (F12) coverage later on as well as help color-blind people. Apart from a coverage ruler, source code markers were implemented so that a user can quickly see and jump to uncovered probes in the same file using an overview ruler built into Eclipse source code editors or using a Problems view, which is also built into Eclipse.

6.4 Experience

Although the initial design was fine, there were several conceptual problems during implementation. To start with, it was not clear how to show annotations when several conditions or decisions occurred on a single line. The first approach was to copy behavior of BullseyeCoverage

Browser, and automatically insert newline characters so that each line contains only one condition or decision. Unfortunately, this did not work because inserting newlines into a file without user consent was difficult to defense and brought a lot of interaction problems. So, the next approach was to put a “mixed state” symbol next to lines, which contain several conditions or decision, and show a pop-up window with coverage information for this line split into several shorter lines. This approach worked.

Implementation of coverage statistics view was easier than implementation of source code annotations, but there also were several smaller problems such as locating user-selected source code element in Eclipse CDT AST and opening it in an editor or such as improving the performance of this view when it contains several thousand items.

Purple Scout's customer representatives liked the prototype demonstrated to them six weeks after the start of development, although it only had some features implemented: F01, F02, F03 (partially), F04, F05, F10 and F13, and met all non-functional requirements. As a consequence, we have soon been offered employment and transferred to other assignments, while the prototype was put in production. On one hand, it was a disappointment to be unable to finish the work and implement all the planned features, but on the other hand the fact that the plug-in was put in production attests to its usefulness. The customer representative has also told us that developers usually rely on continuous integration system to measure code coverage and only if it is lower than a pre-defined threshold, do they perform coverage analysis themselves. This analysis has become much easier with the help of the BullseyePlugin we developed.

It has also been a pleasure to extend Eclipse by writing a plug-in for it. Despite the size of the code base, it is well modularized and easy to extend through the use of variation points. The ability to read its source code has also helped us understand how it works and find usage examples.

Finally, various Eclipse libraries, and especially EMF turned out to be extremely powerful and robust. We got 12000 lines of high quality generated source code in a few days and were amazed that it met requirements N01 and N03 out of box.

The only thing we would have changed if we started from scratch would be to be more strict about unit testing so that it is not postponed until a later day, which may never come.

7. Discussion

7.1 Results

Question 1: What are common coverage criteria and their limits of applicability?
They are function, loop, line, statement, block, decision, condition, condition/decision, multiple condition, modified condition/decision, linear code sequence and jump, path and synchronization coverage. Most of these criteria are only applicable to a few programming languages, although they may be similar counterparts for other languages. Furthermore, even given a single programming language, it is not uncommon to see different interpretations of the same criterion by different tool vendors.

Question 2: Which test suite qualities can be measured or affected by utilizing code coverage data?
Test suite completeness, minimization, prioritization can be measured by utilizing code coverage data. Furthermore, it is plausible that test suite maintainability and modifiability can also be measured and improved through the use of code coverage.

Question 3: What are common ways to present code coverage data and do they faithfully reveal important test suite qualities?
Code coverage data is usually visualized through source code annotation and an aggregate coverage statistics view, which can be used to assess test suite completeness. Using color-coding is very popular for source code annotation, although other methods exist. Aggregate coverage statistics is often represented by a TableTree. Unfortunately, other presentation formats are not widespread, and it is difficult to say if they faithfully reveal other test suite qualities.

Question 4: How to implement an Eclipse plug-in for BullseyeCoverage so that code coverage data can be fully utilized?
On one hand, implementation of a plug-in was originally limited by the choice of a programming language and a coverage tool, so many test suite qualities could not be revealed from the very beginning. On the other hand, while code coverage data is only utilized for assessing test suite completeness by most tools, it is difficult to persuade a customer that he or she needs other features. However, implementation of an Eclipse plug-in itself is quite straightforward.

7.2 Critical analysis

The results presented in the previous chapters have been obtained with the methods described in the beginning of this work. Even though all precautions have been taken to avoid errors, the authors of this paper may have made errors. These errors are discussed below.

The results of this research are largely based on the reviewed literature as well as studied software, and the choice of the literature as well as the choice of software may have affected these results. In order to avoid this problem, the authors have reviewed more literature than mentioned in this paper, and only referenced the most important works. The fact that a lot of literature discovered via different search engines and through referenced work lists, in general agree on the subject suggests

that the literature choice was representative. On the other hand, certain results such as the concept of synchronization coverage, or the use of code coverage data to guide automatic test case generation, are based on relatively few papers and are not widely known. The reliability of such results should be lower, but the authors of this thesis have deemed them to be trustworthy enough to cite in this work.

One potential omission in this work is reliance on analogy and brainstorming during the choice of test case and test suite qualities. However, it is partially mitigated by the fact that the reviewed literature has not contained references to other qualities than the mentioned ones. The choice of the software used for case studies has also been guided by the authors perception of widespread programming languages and the frequency with which different software tools are mentioned. The lists of qualities and studied coverage tools are two areas where the this research could have been improved.

Finally, the fact that implementation of the plugin has never been completed according to initial plans may have affected authors conclusions. Nevertheless, the plugin has been used in production for a while, which can be interpreted as a development success.

7.3 Implications and further work

One of the most interesting observations from this work is the absence of common precise definitions for widespread coverage criteria. The same problem has been spotted by researchers from TU Darmstadt, who have developed an innovative approach to define coverage (Holzer – Schallhart – Tautschnig – Veith 2009). However, their approach relies on a control-flow graph representation of a program, and as such is difficult to apply to functional languages. It would be interesting to see a similar solution for software written in Haskell or Erlang. We believe that the lack of precise definitions can partially explain the lack of quantitative studies measuring effectiveness of using code coverage to assess various qualities of a test suite.

Another interesting result described in the same paper is ability to guide test case generation based on code coverage data. It has potential to revolutionize software testing, but requires more research as well as better programming language definitions. The last point is especially important for designers of future programming languages, who should plan how programs written in these languages will be structured, tested, statically analyzed and how the coverage would be measured. Quite often inability to fully utilize code coverage data comes from limitations of a programming language itself.

Finally, it is a pity that numerous code coverage criteria and methods to utilize coverage data are not implemented in tools. Industry seems to be fixed on the idea of utilizing coverage information exclusively for test suite completeness evaluation, and very few industrial tools go beyond this usage. There is also a need to come up with new ideas about relating coverage information for generated source code to the original code model. For example, this problem exists for C and C++ software, which relies on pre-processor macros as well as for Java software, which relies on dynamic code weaving, e.g. in order to support aspect-oriented programming.

Concerning implementation of a coverage plug-in for Eclipse IDE, it is noteworthy that the choice of the programming language was not as important as the choice of the coverage tool, which has set the boundaries of what we can and cannot do in the plug-in. If we were to do the same project again, we would start by creating and prioritizing the list of features which are feasible given the selected coverage library. For example, one person may be interested in speeding up regression testing based on coverage data, whereas for another person it may be important to easily observe coverage of various source code parts. Understanding which user needs can be addressed by code coverage and which features are feasible would have made the development better focused and more predictable.

8. References

Atlassian 2013. *Clover: Java and Groovy Code Coverage*. Retrieved from https://www.atlassian.com/software/clover/overview on 2013-08-17.

Berenbach, B. – Paulish, D. – Kazmeier, J. – Rudorfer, A. 2009. *Software & Systems Requirements Engineering: In Practice*. United States: McGraw-Hill. ISBN 978-0-07-160547-2.

Bron, A. – Farchi, E. – Magid, Y. – Nir, Y. – Ur, S. 2005. Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 206-212). New York, United States: ACM. doi: 10.1145/1065944.1065972. Retrieved through Chalmers University Library website from http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=1065972 on 2013-08-17.

Bullseye Testing Technology 2013. Measurement Technique. *BullseyeCoverage Introduction.* Retrieved from http://www.bullseye.com/help/introduction.html#2 on 2013-08-17.

Chilenski, J. J. – Miller, S. P. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5). Retrieved through Chalmers University Library website from http://ieeexplore.ieee.org.proxy.lib.chalmers.se/xpl/articleDetails.jsp?arnumber=329068 on 2013-10-09.

Chilenski, J. J. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. United States Federal Aviation Administration report DOT/FAA/AR-01/18. Retrieved from http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf on 2013-08-17.

Clarke, L. A. – Podgurski, A. – Richardson, D. J. – Zeil S. J. 1989. A Formal Evaluation of Data Flow Path Selection. *IEEE Transactions on Software Engineering*, 15(11), 1318 – 1332. doi: 10.1109/32.41326. Retrieved through Chalmers University Library website from http://ieeexplore.ieee.org.proxy.lib.chalmers.se/xpls/abs_all.jsp?arnumber=41326 on 2013-10-13.

Cornett, S. 2011. *Code Coverage Analysis*. Retrieved from http://www.bullseye.com/coverage.html on 2013-08-17.

Dern, C. – Tan, R. P. 2009. Code Coverage for Concurrency. *MSDN Magazine*, 24(9). Retrieved from http://msdn.microsoft.com/en-us/magazine/ee412257.aspx on 2013-08-17.

Dignan, L. 2011. Forrester: Global tech spending to grow 7.1 percent amid software surge. *ZDNet.com Between the Lines blog*, 2011-01-07. Retrieved from http://www.zdnet.com/blog/btl/forrester-global-tech-spending-to-grow-71-percent-amid-software-surge/43379 on 2013-08-17.

Ericsson AB 2012. Cover: A Coverage Analysis Tool for Erlang. *Erlang Tools Reference Manual*, version 2.6.11. Retrieved from http://www.erlang.org/doc/man/cover.html on 2012-10-30.

Frankl P. G. – Iakounenko, O. 1998. Further Empirical Studies of Test Effectiveness. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering* (pp. 153-162). New York, United States: ACM. doi: 10.1145/288195.288298. Retrieved through Chalmers University Library website from http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=288298 on 2013-10-27.

Free Software Foundation, Inc. 2010a. Invoking gcov. *GNU Compiler Collection 4.8.1 Manual*. Retrieved from http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Invoking-Gcov.html on 2013-08-17.

Free Software Foundation, Inc. 2010b. Optimize options. *GNU Compiler Collection 4.8.1 Manual*. Retrieved from http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Optimize-Options.html on 2013-10-09.

Gill, A. – Runciman, C. 2007. Haskell program coverage. In *Proceedings of the 2007 ACM SIGPLAN Haskell workshop* (pp. 1-12). New York, United States: ACM. doi: 10.1145/1291201.1291203. Retrieved through Chalmers University Library website from http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1291201.1291203 on 2013-08-17.

Hayhurst, K. – Veerhusen, D. – Chilenski, J. – Rierson, L. 2001. *A Practical Tutorial on Modified Condition/ Decision Coverage*. United States National Aeronautics and Space Administration report NASA/TM-2001-210876. Retrieved from http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf on 2013-08-17.

Hennel, M. A. – Woodward, M. R. – Hedley, D. 1976. On program analysis. *Information Processing Letters*, 5(5), 136-140. doi: 10.1016/0020-0190(76)90059-4. Retrieved through Chalmers Univesity Library website from http://dx.doi.org.proxy.lib.chalmers.se/10.1016/0020-0190(76)90059-4 on 2013-08-17.

Holzer, A. – Schallhart, C. – Tautschnig, M. – Veith, H. 2009. *A Precise Specification Framework for White Box Program Testing*. Technische Universität Darmstadt report TUD-CS-2009-0148. Retrieved from http://tuprints.ulb.tu-darmstadt.de/id/eprint/1919 on 2013-08-17.

Kegler, J. 2008. Perl Is Undecidable. *The Perl Review*, Fall 2008, 7-11. Retrieved from http://www.jeffreykegler.com/Home/perl-and-undecidability on 2013-08-17.

King, R. S. 2011. The Top 10 Programming Languages. *IEEE Spectrum magazine*, 28 Sep 2011. Retrieved from http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages on 2013-08-17.

Koskela, L. 2004. Introduction to Code Coverage. Basic measures. Java tools. Retrieved from http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html on 2013-08-17.

Mansmann, F – Vinnik, S. 2006. Interactive Exploration of Data Traffic with Hierarchical Network Maps. *IEEE Transactions on Visualization and Computer Graphics*, 12(6), 1440–1449. doi: 10.1109/TVCG.2006.98. Retrieved through Chalmers University Library website from http://dx.doi.org.proxy.lib.chalmers.se/10.1109/TVCG.2006.98 on 2013-08-17.

Marick, B. 1997. How to Misuse Code Coverage. Retrieved from http://www.exampler.com/testing-com/writings/coverage.pdf on 2013-10-27.

Midtgaard, J. – Jensen, T. P. 2012. Control-flow analysis of function calls and returns by abstract interpretation. *Information and Computation*, 211, 49–76. doi: 10.1016/j.ic.2011.11.005. Retrieved through Chalmers University Library website from http://dx.doi.org.proxy.lib.chalmers.se/10.1016/j.ic.2011.11.005 on 2013-08-17.

Miller, T. – Padgham, L. – Thangarajah, J. 2011. Test Coverage Criteria for Agent Interaction Testing. In Weyns, D. – Gleizes, M.-P. (Eds.), *Agent-Oriented Software Engineering XI: 11th International Workshop, AOSE 2010, Toronto, Canada, May 10-11, 2010, Revised Selected Papers* (pp. 91–105). Heidelberg, Germany: Springer Berlin. doi: 10.1007/978-3-642-22636-6_6. Retrieved through Chalmers University Library website from http://dx.doi.org.proxy.lib.chalmers.se/10.1007/978-3-642-22636-6_6 on 2013-08-17.

Mountainminds GmbH & Co. KG and Contributors 2011. Version 2.0.0 (2011/12/18). *EclEmma Change Log*. Retrieved from http://www.eclemma.org/changes.html on 2013-08-18.

Mountainminds GmbH & Co. KG and Contributors 2013. Coverage Counters. *JaCoCo Documentation, 0.6.4.20130617-1803*. Retrieved from http://www.eclemma.org/jacoco/trunk/doc/counters.html on 2013-08-18.

Myers, G. J. 2004a. White-Box Testing. Boundary-Value Analysis. In *The art of software testing* (2nd ed. revised and updated by Tom Badgett and Todd M. Thomas with Corey Sandler) (ch. 4). Hoboken, N.J., United States: Wiley. ISBN 0471469122. Retrieved through Chalmers University Library from http://library.books24x7.com.proxy.lib.chalmers.se/toc.aspx?site=Y7V97&bookid=11258 on 2013-08-18.

Myers, G. J. 2004b. The strategy. In *The art of software testing* (2nd ed. revised and updated by Tom Badgett and Todd M. Thomas with Corey Sandler) (ch. 4). Hoboken, N.J., United States: Wiley. ISBN 0471469122. Retrieved through Chalmers University Library from http://library.books24x7.com.proxy.lib.chalmers.se/toc.aspx?site=Y7V97&bookid=11258 on 2013-08-18.

Nakkhongkham, S. 2011. *Assessing the Relationship between Prerelease Software Testing and the Number of Product Defects Discovered*. PhD. Northcentral University, USA. Ann Arbor, US: UMI Dissertations Publishing.

Ormandy, T. 2009. Making software dumber. Retrieved from http://taviso.decsystem.org/making_software_dumber.pdf on 2013-09-20.

RTI International 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. United States National Institute of Standards and Technology Planning Report 02-3. Retrieved from http://www.nist.gov/director/planning/upload/report02-3.pdf on 2013-08-18.

Rothermel, G. – Untch, R. H. – Chu, C. – Harrold, M. J. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, 27(10), 929-948. doi: 10.1109/32.962562. Retrieved through Chalmers University Library from http://ieeexplore.ieee.org.proxy.lib.chalmers.se/xpls/abs_all.jsp?arnumber=962562 on 2013-10-31.

Roubtsov, V. 2006. What are those yellow lines in the HTML report? *EMMA: Frequently Asked Questions*. Retrieved from http://emma.sourceforge.net/faq.html#N102B6 on 2013-09-01.

Swedsoft 2010. Mjukvaran är själen i svensk industri (in Swedish). Retrieved from http://www.swedsoft.se/Mjukvaran_%C3%A4r_sj%C3%A4len_i_svensk_industri.pdf on 2012-05-07.

Spear, C. – Tumbush, G. 2012. Functional Coverage. In *SystemVerilog for Verification* (pp. 323-361). United States: Springer. Retrieved through Chalmers University Library from http://dx.doi.org.proxy.lib.chalmers.se/10.1007/978-1-4614-0715-7_9 on 2013-09-14.

Testwell 2013a. Testwell CTC++ Description. Retrieved from http://www.testwell.fi/ctcdesc.html on 2013-09-01.

Testwell 2013b. CTC++ Coverage Report – Execution Profile. Retrieved from http://www.testwell.fi/CTCHTML/indexA.html on 2013-09-01.

The GHC Team 2007. Release notes for version 6.8.1. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.8.1*. Retrieved from http://www.haskell.org/ghc/docs/6.8.1/html/users_guide/release-6-8-1.html on 2013-09-01.

Troster, J. 1992. Assessing design-quality metrics on legacy software. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, vol. 1* (pp. 113-131). Toronto, Canada: IBM Press. Retrieved through Chalmers University Library from http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=962209 on 2013-10-31.

Widera, M. 2005. Data Flow Coverage for Testing Erlang Programs. In *Proceedings of the Sixth Symposium on Trends in Functional Programming* (pp. 151-166). Tallinn, Estonia: Institute of Cybernetics. Retrieved from http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/ on 2013-09-01.

Williams, T. W. – Mercer, M. R. – Mucha, J. P. – Kapur, R. 2001. Code coverage, what does it mean in terms of quality? In *Proceedings of the 2001 Annual Reliability and Maintainability Symposium* (pp. 420-424). doi: 10.1109/RAMS.2001.902502. USA: IEEE Computer Society.

Wong, W. E. – Horgan, J. R. – London, S. – Mathur, A.P. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *Proceedings of the 17th international conference on Software engineering*, pp. 41-50. doi: 10.1145/225014.225018. New York, USA: ACM. Retrieved through Chalmers University Library from http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=225018 on 2013-10-31.

Woodward, M. R. – Hennel, M. A. – Hedley, D. 1980. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, 6(3), 278-286. doi: 10.1109/TSE.1980.230473. Retrieved through Chalmers University Library from http://dx.doi.org.proxy.lib.chalmers.se/10.1109/TSE.1980.230473 on 2013-09-01.

Woodward, M. R. – Hennel, M. A. 2006. On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC. *Information and Software Technology*, 48(7), 433-440. doi: 10.1016/j.infsof.2005.05.003. Retrieved through Chalmers University Library from http://dx.doi.org.proxy.lib.chalmers.se/10.1016/j.infsof.2005.05.003 on 2013-09-01.

Zhu, H. – Hall, P. A. – May, J. H. 1997. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4), 366 – 427. doi: 10.1145/267580.267590. Retrieved through Chalmers University Library from http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=267590 on 2013-09-01.