# Static Estimation of Test Coverage

Tiago L. Alves
*Universidade do Minho, Portugal, and*
*Software Improvement Group, The Netherlands*
*Email: t.alves@sig.nl*

Joost Visser
*Software Improvement Group*
*The Netherlands*
*Email: j.visser@sig.nl*

***Abstract*—Test coverage is an important indicator for unit test quality. Tools such as Clover compute coverage by first instrumenting the code with logging functionality, and then logging which parts are executed during unit test runs.**

**Since computation of test coverage is a dynamic analysis, it presupposes a working installation of the software. In the context of software quality assessment by an independent third party, a working installation is often not available. The evaluator may not have access to the required libraries or hardware platform. The installation procedure may not be automated or documented.**

**In this paper, we propose a technique for estimating test coverage at method level through static analysis only. The technique uses slicing of static call graphs to estimate the dynamic test coverage. We explain the technique and its implementation. We validate the results of the static estimation by statistical comparison to values obtained through dynamic analysis using Clover. We found high correlation between static coverage estimation and real coverage at system level but closer analysis on package and class level reveals opportunities for further improvement.**

## I. Introduction

In the object-oriented community, *unit testing* is a white-box testing method for developers to validate the correct functioning of the smallest testable parts of source code [1]. Object-oriented unit testing has received broad attention and enjoys increasing popularity, also in industry [2].

A range of frameworks has become available to support unit testing, including SUnit, JUnit, and NUnit[1]. These frameworks allow developers to specify unit tests in source code and run suites of tests during the development cycle.

A commonly used indicator to monitor the quality of unit tests is *code coverage*. This notion refers to the portion of a software application that is actually used during a particular execution run. The coverage obtained when running a particular suite of tests can be used as an indicator of the quality of the test suite and, by extension, of the quality of the software if the test suite is passed successfully.

Tools are available to compute code coverage during test runs [3]. These tools work by instrumenting the code with logging functionality before execution. The logging information collected during execution is then aggregated and reported. For example, Clover[2] instruments Java source code and reports statement coverage and branch coverage at the level of methods, classes, packages, and the overall system. Emma[3] instruments Java bytecode, and reports statement coverage and method coverage at the same levels. The detailed reports of such tools provide valuable input to increase or maintain the quality of test code.

Computing code coverage involves running the application code and hence requires a working installation of the software. In the context of software development, satisfaction of this requirement does not pose any new challenge.

However, in other contexts this requirement can be highly impractical or impossible to satisfy. For example, when an independent party evaluates the quality and inherent risks of a software system [4], [5], there are several compelling reasons that put availability of a working installation out of reach. The software may require hardware not available to the assessor. The build and deployment process may not be reproducible due to a lack of automation or documentation. The software may require proprietary libraries under a non-transferrable license. In embedded software, for instance, instrumented applications by coverage tools may not run or may display altered behavior due to space or performance changes. Finally, for a very large system it might be too expensive to frequently execute the complete test suite, and subsequently, compute coverage reports.

These limitations derive from industrial practice in analyzing software that may be incomplete and may not be possible to execute. To overcome these limitations a light-weight technique to estimate test coverage previous to running the test cases is necessary.

The question that naturally arises is: could code coverage by tests possibly be determined without actually running the tests? And which trade-off may be made between sophistication of such a static analysis and its accuracy?

In this paper we answer these questions. We propose a static analysis for estimating code coverage, based on slicing of call graphs (Section II). We discuss the sources of imprecision inherent in this analysis as well as the impact of imprecision on the results (Section III). We experimentally assess the quality of the static estimates compared to the dynamically determined code coverage results for a range of

[1]sunit.sourceforge.net, www.junit.org, www.nunit.org
[2]http://www.atlassian.com/software/clover/
[3]http://emma.sourceforge.net/

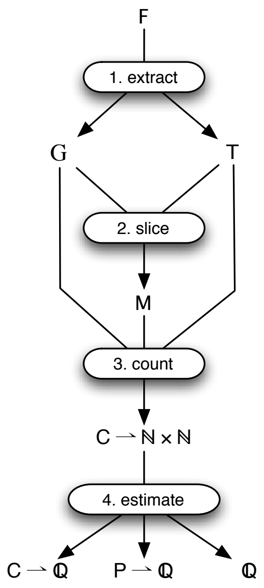
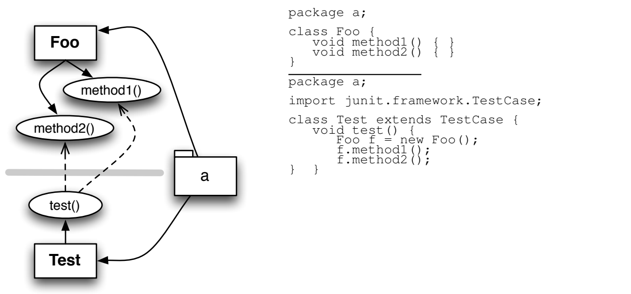


Figure 1. Overview of the approach. The input is a set of files ($F$). From these files, a call graph is constructed ($G$). Also, the test classes of the system are identified ($T$). Slicing is performed on the graph with the identified test classes as entry points, and the production code methods ($M$) in the resulting slice are collected. These covered methods allow to count for each class in the graph (i) how many methods it defines (ii) how many of these are covered. Finally, the estimations of coverage ratio's are then computed on the class, package and system levels. The harpoon arrow denotes a map.



```
package a;

class Foo {
   void method1() { }
   void method2() { }
}
```

```
package a;

import junit.framework.TestCase;

class Test extends TestCase {
   void test() {
      Foo f = new Foo();
      f.method1();
      f.method2();
}  }
```

Figure 2. Source code fragment and the corresponding graph structure, showing different types of nodes (package, class and method) and edges (class and method definition and method calls).

proprietary and open source software systems (Section IV). We discuss related work in Section V and we conclude with a summary of contributions and future work in Section VI.

## II. Approach

Our approach for estimating code coverage involves reachability analysis on a graph structure (graph slicing [6]). This graph is obtained from source code via static analysis. The granularity of the graph is at the method level, and control or data flow information is not assumed. Test coverage is estimated by calculating the ratio between the number of production code methods reached from tests and the overall number of production code methods.

An overview of the various steps of the analysis is given in Figure 1. We briefly enumerate the steps before explaining them in detail in the upcoming sections:

1) From all source files $F$, including both production and test code, a graph $G$ is extracted via static analysis which records both structural information and call information. Also, the test classes are collected in a set $T$.
2) From the set of test classes $T$, we determine test methods and use them as slicing criteria. From test method nodes, the graph $G$ is sliced, primarily along call edges, to collect in a set $M$ all methods reached.
3) For each production class in the graph, the number of methods defined in that class is counted. Also the set of covered methods is used to arrive at a count of covered methods in that class. This is depicted in Figure 1 as a map from classes to a pair of numbers.
4) The final estimates at class, package, and system levels are obtained as ratios from the counts per class.

Note that the main difference between the steps of our approach and dynamic analysis tools, like Clover, is in step 2. Instead of using precise information recorded by logging the methods that are executed, we use an estimation of the methods that are called, determined via static analysis. Moreover, while some dynamic analysis tools take test results into account, our approach does not.

The proposed approach is designed with a number of desirable characteristics in mind: only static analysis is used; the graph contains call information extracted from source code; scalable to large systems; granularity is limited to the method level to keep whole-system analysis tractable; robust against partial availability of source code; finally, missing information is not blocking, though it may lead to less accurate estimates. The extent to which these properties are realized will become clear in Section IV. First, we will explain the various steps of our approach in more detail.

*1) Graph construction:* Through static analysis, a graph is derived representing packages, classes, interfaces, and methods, as well as various relations between them. An example is provided in Figure 2. Below, we will explain the node and edge types that are present in such graphs. We relied on the Java source code extraction of the SemmleCode tool [7]. In this section we will provide a discussion of the required functionality, independent of that implementation.

A directed graph can be represented by a pair $G = (V, E)$, where $V$ is the set of vertices (nodes) and $E$ is the set of edges between these vertices. We distinguish four types of vertices, corresponding to packages ($P$), classes ($C$), interfaces ($I$), and methods ($M$). Thus, the set $V$ of vertices can be partitioned into four subsets which we will write as $N_n \in V$ where node type $n \in \{P, C, I, M\}$. In the various figures in this paper, we will represent packages

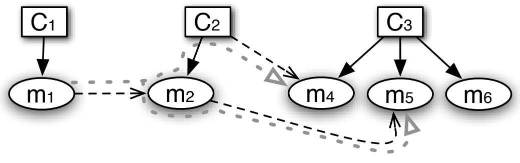


Figure 3. Modified graph slicing algorithm in which calls are taken into account originating from both methods and object initializers. Black arrows represent edges determined via static analysis, and grey arrows depicts the slicing traversal. Full lines are used for method definitions and dashed lines are used for method calls.

as folder icons, classes and interfaces as rectangles, and methods as ellipses. The set of nodes that represent classes, interfaces, and methods is also partitioned to differentiate between production ($PC$) and test code ($TC$). We write $N_n^c$ where code type $c \in \{PC, TC\}$. In the various figures we show production code above a gray separation line and test code below. The edges in the extracted graph structure represent both structural and call information. For structural information two types of edges are used: The *defines type* edges (DT) express that a package contains a class or an interface; the *defines method* edges (DM) express that a class or interface defines a method. For call information, two types of edges are used: *direct call* and *virtual call*. A *direct call* edge (DC) represent a method invocation. The origin of the call is typically a method but can also be a class or interface in case of method invocation in initializers. The target of the call edge is the method definition to which the method invocation can be *statically* resolved. A *virtual call* edge (VC) is constructed between a caller and any implementation of the called method that it might be resolved to during runtime, due to dynamic dispatch. An example will be shown in Section III-A. The set of edges is actually a relation between vertices such that $E \subseteq \{(u,v)_e \,|\, u, v \in V\}$, where $e \in \{DT, DM, DC, VC\}$. We write $E_e$ for the four partitions of $E$ according to the various edges types. In the figures, we will use solid arrows to depict *defines* edges and dashed arrows to depict *calls*. Further explanation of the two types of call edges are provided in Section III-A.

*2) Identifying test classes:* Several techniques can be used to identify test code. We have investigated the possibility to statically determine test code by recognizing the use of a known test library, such as JUnit. A class is considered as a test class if uses the testing library. Although this technique is completely automatic, it fails to recognize test helper classes, i.e., classes with the single purpose of easing the process of testing, which do not need to have any reference to a test framework. Alternatively, naming conventions are used to determine test code. We observed that the majority of proprietary and open-source systems store production and test code in different file system paths. The only drawback of this technique is that, for each system, this path must be manually determined since each project uses its own naming convention.

*3) Slicing to collect covered methods:* In the second step, graph slicing [6] is applied to collect all methods covered by tests. We use the identified set of test classes and their methods as slicing criteria (starting points). We then follow the various kinds of *call* edges in forward direction to reach all covered methods. In addition, we refined the slicing algorithm to take into account *call* edges originating in the object initializers. The modification consists in following *define method* edges backward from covered methods to their defining classes, which then triggers subsequent traversal to the methods invoked by the initializers of those classes. Our slicing algorithm is depicted in Figure 3. The edge from $C_2$ to $m_4$ exemplifies an initializer call.

The modified slicing algorithm can be defined as follows. We write $n \xrightarrow{call} m$ for an edge in the graph that represents a node $n$ calling a method $m$, where the call type can be vanilla or virtual. We write $m \xleftarrow{def} c$ for the inverse of a *define method* edge, i.e., to denote a function that returns the class $c$ in which a method $m$ is defined. We write $n \xrightarrow{init} m$ for $n \xrightarrow{call} m_i \xleftarrow{def} c \xrightarrow{call} m$, i.e., to denote that a method $m$ is reached from a node $n$ via a class initialization triggered by a call to method $m_i$ (e.g., $m_1 \xrightarrow{init} m_4$, in which $m_1 \xrightarrow{call} m_2 \xleftarrow{def} C_2 \xrightarrow{call} m_4$). Finally, we write $n \xrightarrow{invoke} m$ for $n \xrightarrow{call} m$ or $n \xrightarrow{init} m$. Now, let $n$ be a graph node corresponding to a class, interface or a method (i.e. package nodes are not considered). Then, a method $m$ is said to be reachable from a node $n$ if $n \xrightarrow{invoke}^{+} m$.

These declarative definitions can be encoded in a graph traversal algorithm. Our implementation, however, was done using the relational query language .QL [7].

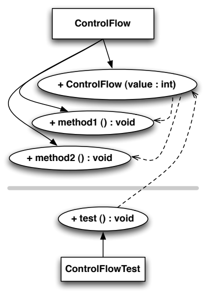
*4) Count methods per class:* In the third step we compute the two core metrics for the static test coverage estimation:

- **Number of defined methods per class (DM)**, defined as $DM : n_C \rightarrow \mathbb{N}$. This metric is calculated by counting the number of outgoing *define method* edges per class.
- **Number of covered methods per class (CM)**, defined as $CM : n_C \rightarrow \mathbb{N}$. This metric is calculated by counting the number of outgoing *define method* edges where the target is a method contained in the set of covered methods.

These statically computed metrics are stored in a finite map $n_C \rightharpoonup \mathbb{N} \times \mathbb{N}$. This map will be used to compute coverage at class, package and system levels as shown below.

*5) Estimate static test coverage:* After computing the two basic metrics we can obtain derived metrics: coverage per class, packages, and system.

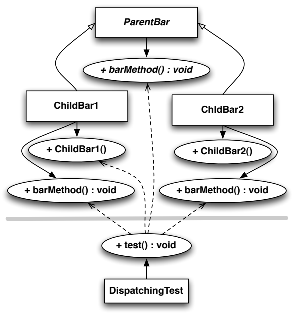
- **Class coverage** Method coverage at the class level is the ratio between covered and defined methods per class:
$$CC(c) = \frac{CM(c)}{DM(c)} \times 100\%$$
- **Package coverage** Method coverage at the package level is the ratio between the total number of covered



```
class ControlFlow {
  ControlFlow(int value) {
     if (value > 0)
        method1();
     else
        method2();
  }
  void method1() { }
  void method2() { }
}
```

```
import junit.framework.*;

class ControlFlowTest
      extends TestCase {
   void test() {
      ControlFlow cf =
         new ControlFlow(3);
}  }
```

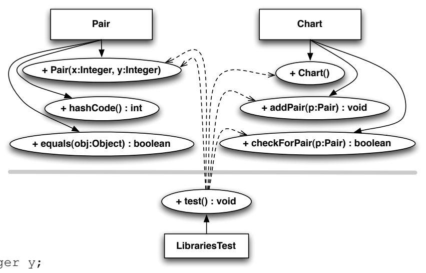
Figure 4. Imprecision related to control flow.



```
class ParentBar {
   void barMethod() { };
}

class ChildBar1
      extends ParentBar {
   ChildBar1() { }

   void barMethod() { }
}

class ChildBar2
      extends ParentBar {
   ChildBar2() { }

   void barMethod() { }
}
```

```
import junit.framework.*;

class DynamicDispatchTest
      extends TestCase {
   void test() {
      ParentBar p =
         new ChildBar1();
      p.barMethod();
}  }
```

Figure 5. Imprecision: dynamic dispatch.


Pair
Chart
+ Pair(x:Integer, y:Integer)
+ Chart()
+ hashCode() : int
+ addPair(p:Pair) : void
+ equals(obj:Object) : boolean
+ checkForPair(p:Pair) : boolean
+ test() : void
LibrariesTest

```
class Pair {
   Integer x; Integer y;

   Pair(Integer x, Integer y) {
    ...
   }
   int hashCode() { ... }
   boolean equals(Object obj) {
      ...
}  }
class Chart {
   Set pairs;

   Chart() { pairs = new HashSet(); }

   void addPair(Pair p) {
      pairs.add(p);
   }
   void checkForPair(Pair p) {
      return pairs.contains(p);
}  }
```

```
import junit.framework.*;

class LibrariesTest
      extends TestCase {

   void test() {
      Chart c = new Chart();

      Pair p1 = new Pair(3,5);
      c.addPair(p1);

      Pair p2 = new Pair(3,5);
      c.checkForPair(p2);
}  }
```

Figure 6. Imprecision: library calls.

methods and the total number of defined methods per package:

$$PC(p) = \frac{\sum_{c \in p} CM(c)}{\sum_{c \in p} DM(c)} \times 100\%$$

where $c \in p$ iff $c \in V^P \wedge c \xleftarrow{def} p$, i.e., $c$ is a production class in package $p$.

- **System coverage** Method coverage at system level is the ratio between the total number of covered methods and the total number of defined methods in the overall system:

$$SC = \frac{\sum_{c \in G} \mathit{CM(c)}}{\sum_{c \in G} \mathit{DM(c)}} \times 100\%$$

where $c \in G$ iff $c \in V^P$, i.e., $c$ is a production class.

## III. Imprecision

Our coverage analysis is based on a statically derived graph, in which the structural information is exact and the method call information is an approximation of the dynamic execution. The precision of our estimation is a function of the precision of the call graph extraction. As mentioned before, we rely here on SemmleCode. Below we will discuss the sources of imprecision independent of the tool.

In this section we discuss various sources of imprecision: control flow, dynamic dispatch, framework/library calls, identification of production and test code, and failing tests. We also discuss how to deal with the imprecision.

### A. Sources of imprecision

*Control flow:* Figure 4 presents an example where the graph contains imprecision due to control flow, i.e., due to the occurrence of method calls under conditional statements. In this example, if `value` is greater than zero `method1` is called, otherwise `method2` is called. In the test, the value 3 is passed as argument and `method1` is called. However, without data flow analysis or partial evaluation, it is not possible to statically determine which branch is taken, and which methods are called. For now we will consider an optimistic estimation, considering both `method1` and `method2` calls. Further explanations about how to deal with imprecision are given in Section III-B.

Other types of control-flow statements will likewise lead to imprecision in our call graphs, such as `switch` statements, looping statements (`for`, `while`), and branching statements (`break`, `continue`, `return`).

*Dynamic dispatch:* Figure 5 presents an example of imprecision due to dynamic dispatch. A parent class `ParentBar` defines `barMethod`, which is redefined by two subclasses (`ChildBar1` and `ChildBar2`). In the test, a `ChildBar1` object is assigned to a variable of the `ParentBar` type, and the `barMethod` is invoked. During test execution, the `barMethod` of `ChildBar1` is called. Our static analysis, however, identifies all three implementations of `barMethod` as potential call targets, represented in the graph as three edges: one *direct call* edge to the `ParentBar` and two *virtual call* edges to the `ParentChild1` and `ParentChild2` implementations.

*Frameworks / Libraries:* Figure 6 presents an example of imprecision caused by frameworks/library calls of which no code is available for analysis. The class `Pair` represents a two-dimensional coordinate, and class `Chart` contains all the points of a chart. `Pair` defines a constructor and redefines the `equals` and `hashCode` methods to enable the comparison of two objects of the same type. In the test, a `Chart` object is created and the coordinate $(3, 5)$ is added to the chart. Then another object with the same coordinates is created and checked to exist in the chart. When a `Pair` object is added to the set, and when checking if an object exists in a set, the methods `hashCode` and `equals` are called. These calls are not present in our call graph.

*Identification of production and test code:* Failing to distinguish production from test code has a direct impact on the coverage estimation. Recognizing tests as production code increases the size of the overall production code and hides calls from tests to production code, causing a decrease of coverage (underestimation). Recognizing production code as tests, has the opposite effect, decreases the size of overall production code and increases the number of calls causing an increase of coverage (overestimation).

As previously stated, the distinction between production and test code is done using file system path information. A class is considered test code if it is inside a test folder, and considered production code if it is inside a non-test folder. Since most projects and tools (e.g., Clover) respect and use this convention we consider this approach safe.

*Failing tests:* Unit testing is done to detect faults which requires assertion of the state and/or results of a unit of code. If the test succeeds the unit under test is considered as test covered. However, if the test does not succeed two alternatives are possible. First, the unit test is considered as not covered, but the unit under test is considered as test covered. Second, both the unit test and the unit under test are considered as not test covered. Emma is an example of the first case, while Clover is an example of the second.

Failing tests can cause imprecision since Clover will consider the functionality under test as not covered while our approach, which is not sensitive to test results, will consider the same functionality as test covered. However, failing tests are not common in released software.

### B. *Dealing with imprecision*

Of all sources of imprecision, we are concerned with control flow, dynamic dispatch and frameworks/libraries only. Imprecision caused by test code identification and failing tests will not be considered, since are less common.

To deal with imprecision without resorting to detailed control and data flow analyses, two approaches are possible. In the pessimistic approach, we only follow call edges that are guaranteed to be exercised during execution. This will result in a systematic underestimation of test coverage. In an optimistic approach, all potential call edges are followed, also if they are not necessarily exercised during execution. With this approach, test coverage will be overestimated.

In the particular context of quality and risk assessment, only the optimistic approach is suitable. A pessimistic approach would lead to a large number of false negatives (methods that are erroneously reported to be not covered). In the optimistic approach, methods reported to be not covered are with high certainty indeed not covered. Since the purpose is to detect the lack of coverage, only the optimistic approach makes sense. Hence, we will only report values for the optimistic approach. However, the optimistic approach is not consistently optimistic for all the uncertainties previously mentioned: imprecision due to library/framework will always cause underestimation. This underestimation can influence coverage estimation to values lower than Clover coverage. Nevertheless, if a particular functionality is only reached via frameworks/libraries, i.e., it can not be statically reached from a unit test, then it is fair to assume that this functionality is not unit test covered (albeit is considered covered by a test).

In the sequel, we will investigate experimentally the consequences of these choices for the accuracy of the analysis.

## IV. Experimentation

We experimentally compared the results of static estimation of coverage against dynamically computed coverage for several software systems. We did an additional comparison for several revisions of the same software system in order to investigate if our static estimation technique is sensitive to coverage fluctuations.

Although we are primarily interested in coverage at system level, we additionally report coverage at package and class levels.

### A. *Experimental design*

*Systems analyzed:* We analyzed twelve Java systems, ranging from 2.5k to 268k LOC, with a total of 840k LOC (production and test code). The systems are listed in Table I, as well as metric information for production and test code. The list is diverse both in terms of size and scope. JPacMan is a tiny system developed for education. Dom4j, JGAP, Collections and JFreeChart are open-source libraries and PMD is an open-source tool[4]. G System and R System are anonymized proprietary systems. Certification, Analyses and DocGen are proprietary tools and Utils is a proprietary library.

Additionally, we analyzed 52 releases of the Utils project with a total of over 1.2M LOC, with sizes ranging from 4.5k LOC, for version 1.0, to 37.7k LOC, for version 1.61.

For all the analyzed systems, production and test code was distinguishable by file path and no failing tests existed.

*Measurement:* For the dynamic coverage measurement, we use XML reports produced by the Clover tool. XSLT transformations are used to extract the required information.

For the static estimation of coverage, we implemented the extract, slice, and count steps of our approach in a number of relational .QL queries for the SemmleCode tool [7].

[4]http://www.dom4j.org, http://jgap.sourceforge.net, http://commons.apache.org/collections, http://www.jfree.org/jfreechart, http://pmd.sourceforge.net

Table I
CHARACTERIZATION OF THE SYSTEMS USED IN THE EXPERIMENT ORDERED BY LOC.

| System name | Version | Author / Owner | Description | LOC | | #Packages | | #Classes | | #Methods | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Production | Test | Production | Test | Production | Test | Production | Test |
| JPacMan | 3.04 | Arie van Deursen | Game used for OOP education | 1,539 | 960 | 2 | 3 | 29 | 17 | 223 | 112 |
| Certification | 20080731 | SIG | Tool for software quality rating | 2,220 | 1,563 | 14 | 9 | 71 | 28 | 256 | 157 |
| G System | 20080214 | C Company | Database synchronization tool | 3,459 | 2,910 | 15 | 16 | 56 | 70 | 504 | 285 |
| Dom4j | 1.6.1 | MetaStuff | Library for XML processing | 18,305 | 5,996 | 14 | 11 | 166 | 105 | 2921 | 685 |
| Utils | 1.61 | SIG | Toolkit for static code analysis | 20,851 | 16,887 | 37 | 32 | 323 | 183 | 3243 | 1290 |
| JGAP | 3.3.3 | Klaus Meffert | Library of Java genetic algorithms | 23,579 | 19,340 | 25 | 20 | 267 | 184 | 2990 | 2005 |
| Collections | 3.2.1 | Apache | Library of data structures | 26,323 | 29,075 | 12 | 11 | 422 | 292 | 4098 | 2876 |
| PMD | 5.0b6340 | Xavier Le Vourch | Java static code analyzer | 51,427 | 11,546 | 66 | 44 | 688 | 206 | 5508 | 1348 |
| R System | 20080214 | C Company | System for contracts management | 48,256 | 34,079 | 62 | 55 | 623 | 353 | 8433 | 2662 |
| JFreeChart | 1.0.10 | JFree | Java chart library | 83,038 | 44,634 | 36 | 24 | 476 | 399 | 7660 | 3020 |
| DocGen | r40981 | SIG | Cobol documentation generator | 73,152 | 54,576 | 111 | 85 | 1359 | 427 | 11442 | 3467 |
| Analyses | 1.39 | SIG | Tools for static code analysis | 131,476 | 136,066 | 278 | 234 | 1897 | 1302 | 13886 | 8429 |

Table II
STATIC AND CLOVER COVERAGE, AND COVERAGE DIFFERENCES AT SYSTEM LEVEL.

| System name | Static | Clover | Differences |
|---|---|---|---|
| JPacMan | 88.06% | 93.53% | −5.47% |
| Certification | 92.82% | 90.09% | 2.73% |
| G System | 89.61% | 94.81% | −5.19% |
| Dom4j | 57.40% | 39.37% | 18.03% |
| Utils | 74.95% | 70.47% | 4.48% |
| JGAP | 70.51% | 50.99% | 19.52% |
| Collections | 82.62% | 78.39% | 4.23% |
| PMD | 80.10% | 70.76% | 9.34% |
| R System | 65.10% | 72.65% | −7.55% |
| JFreeChart | 69.88% | 61.55% | 8.33% |
| DocGen | 79.92% | 69.08% | 10.84% |
| Analyses | 71.74% | 88.23% | −16.49% |

*Statistical analysis:* For statistical analysis we used R [8]. We created histograms to inspect the distribution of the estimate (static) coverage and the true (Clover) coverage. To visually compare these distributions, we created scatter plots of one against the other, and we created histograms of their differences. To inspect central tendency and dispersion of the true and estimated coverage as well as of their differences, we used descriptive statistics, such as median and interquartile range. To investigate correlation of true and estimated coverage we used a non-parametric method (Spearman's rank correlation coefficient [9]) and a parametric method (Pearson product-moment correlation coefficient). Spearman is used when no assumptions about data distributions can be made. Pearson, on the other hand, is more precise than Spearman, but can only be used if data is assumed to be normal. For testing data normality we use the Anderson-Darling test [10] for a 5% significance level. The null hypothesis of the test states that the data can be assumed to follow the normal distribution, while the alternative hypothesis states that the data cannot be assumed to follow a normal distribution. We reject the null hypothesis for a computed $p$-value smaller or equal than $0.05$.

## B. Experiment results

We discuss the results of the experiment, first at the level of complete systems. Second, we look at several releases of the same project. Third, we look at the class and package level results. Finally, we look at one system in more detail.

*System coverage results:* Table II and the scatter plot in Figure 7 show the estimated (static) and the true (Clover) system-level coverage for all systems. Each dot in Figure 7 represents a system. Table II shows that the differences range from $-16.5$ to $19.5$ percent points. In percent points, the average of absolute differences is $9.35$ and the average difference is $3.57$. Figure 7 shows that static coverage values are close to the diagonal, which depicts the true coverage. For one third of the systems coverage was underestimated while two thirds was overestimated.

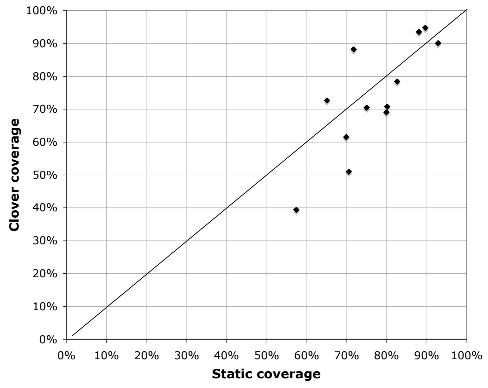


Figure 7. Scatter plot comparing static and dynamic coverage for each system.

Assuming no distribution about data normality, Spearman correlation can be used. Spearman correlation reports $0.769$ with high significance ($p$-value $< 0.01$). Using Anderson-Darling test for data normality, the $p$-values are $0.920$ and $0.522$ for static coverage and clover coverage, respectively. Since, we cannot reject the null hypothesis, i.e., we cannot reject that the data does not belong to a normal distribution, we can use a more accurate method for correlation: Pearson. Pearson correlation reports $0.802$ and $p$-value $< 0.01$. Hence, static and clover coverage are highly correlated with high significance.

*Coverage comparison for the Utils project releases:* Figure 8 plots a comparison between static and dynamic coverage for 52 releases of Utils, from releases 1.0 to 1.61 (some releases were skipped due to compilation problems).

Static coverage is consistently higher than Clover coverage. Despite this overestimation, static coverage follows the same variations as reported by Clover which indicates that

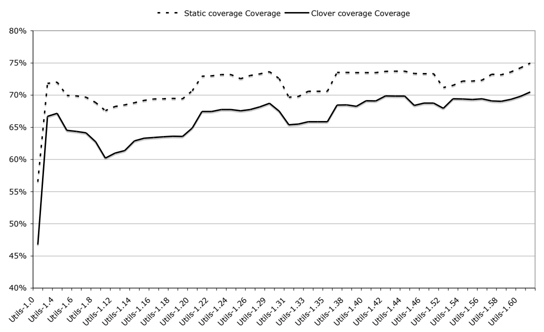


Figure 8. Plot comparing static and dynamic coverage for 52 releases of Utils.

Table III
STATISTICAL ANALYSIS REPORTING CORRELATION BETWEEN STATIC AND CLOVER COVERAGE, AND MEDIAN AND INTERQUARTILE RANGES (IQR) FOR COVERAGE DIFFERENCES AT SYSTEM LEVEL.

| System | Spearman | | Median | | IQR | |
|---|---|---|---|---|---|---|
| name | Class | Package | Class | Package | Class | Package |
| JPacMan | $0.467^{*}$ | $1$ | $0$ | $-0.130$ | $0.037$ | - |
| Certification | $0.368^{**}$ | $0.520$ | $0$ | $0$ | $0$ | $0.015$ |
| G System | $0.774^{**}$ | $0.694^{**}$ | $0$ | $0$ | $0$ | $0.045$ |
| Dom4j | $0.584^{**}$ | $0.620^{*}$ | $0.167$ | $0.118$ | $0.333$ | $0.220$ |
| Utils | $0.825^{**}$ | $0.778^{**}$ | $0$ | $0.014$ | $0$ | $0.100$ |
| JGAP | $0.733^{**}$ | $0.786^{**}$ | $0$ | $0$ | $0.433$ | $0.125$ |
| Collections | $0.549^{**}$ | $0.776^{**}$ | $0$ | $0.049$ | $0.027$ | $0.062$ |
| PMD | $0.638^{**}$ | $0.655^{**}$ | $0$ | $0.058$ | $0.097$ | $0.166$ |
| R System | $0.727^{**}$ | $0.723^{**}$ | $0$ | $-0.079$ | $0.043$ | $0.162$ |
| JFreeChart | $0.632^{**}$ | $0.694^{**}$ | $0$ | $0.048$ | $0.175$ | $0.172$ |
| DocGen | $0.397^{**}$ | $0.459^{**}$ | $0$ | $0.100$ | $0.400$ | $0.386$ |
| Analyses | $0.391^{**}$ | $0.486^{**}$ | $0$ | $-0.016$ | $0.333$ | $0.316$ |

static coverage is able to detect coverage fluctuations.

The application of Anderson-Darling test rejects the null hypothesis for 5% of significance. Hence correlation can only be computed with Spearman. Spearman correlation reports a value of $0.888$ with high significance ($p$-value $< 0.01$), reinforcing that estimated and true system-level coverage are highly correlated with high significance.

*Package and Class coverage results:* Despite the encouraging system-level results, it is also important, and interesting, to analyze the results at package and class levels.

Table III reports for each system the Spearman correlation and significance, and the median (central tendency) and interquartile range (IQR) of the differences between estimated and true values. Correlation significance is depicted without a star for $p$-value $\geq 0.05$, meaning not significant, with a single star for $p$-value $< 0.05$, meaning significant, and two stars for $p$-value $< 0.01$, meaning highly significant. Since Anderson-Darling test rejected that the data belongs to a normal distribution, Spearman correlation was used.

At class level, except for JPacMan (due to its small size), all systems report high significance. The correlation however, varies from $0.368$ (low correlation) for Certification, to $0.825$ (high correlation) for Utils. Spearman correlation, for all systems, reports a moderate correlation value of $0.503$ with high significance ($p$-value $< 0.01$). With respect to the median, all systems are centered in zero, except for Dom4j in which there is a slight overestimation. The IQR is not uniform among systems varying from extremely low values (Collections) to relatively high values (JGAP and DocGen).

At package level, except for JPacMan, Certification and Dom4j, all systems report high significance. Dom4j correlation is significant at $0.05$ level while for Certification and Dom4j is not significant. These low significance levels are due to the small number of packages. Correlation, again, varies from $0.459$ (moderate correlation) for DocGen, to $0.786$ (high correlation) for JGAP. The correlation value for JPacMan is not taken into account since it is not significant. Spearman correlation, for all systems, reports a moderate correlation value of $0.536$ with high significance ($p$-value $< 0.01$). Regarding the median, and in contrast to what was observed for class level, only three systems reported a value of $0$. However, except for JPacMan, Dom4j and DocGen all other systems report values very close to zero. The IQR show more homogeneous values than at class level, with values below $0.17$ except for JPacMan (which does not have IQR due to its sample size of 3) and Dom4j, DocGen and Analyses whose values are higher than $0.20$.

We now discuss the results for the Collections project. For other systems the conclusions would be similar.

*Collections system analysis:* Figure 9 shows two histograms for the distributions of estimated (static) and true (Clover) class-level coverage for the Collections library. The figure reveals that static coverage was accurate to estimate true coverage in all ranges, with minor oscillations in the

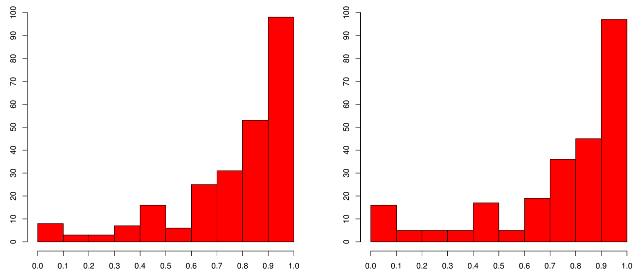


Figure 9. Histograms of static and Clover class coverage for Collections, respectively.

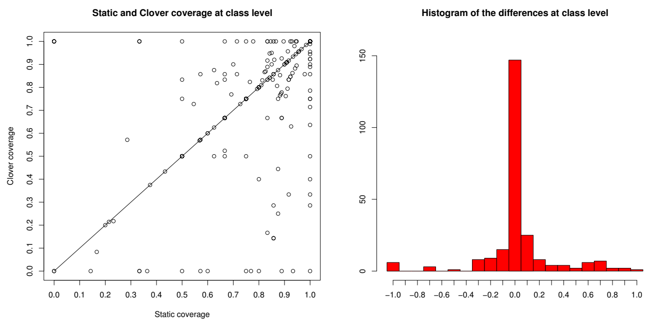


Figure 10. Scatter of static and Clover coverage and histogram of the coverage differences for Collections at class level.

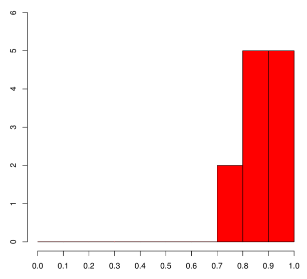
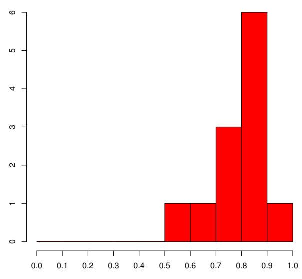
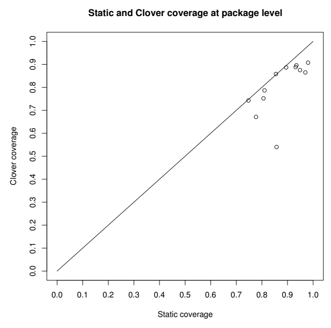



Figure 11. Histograms of static and Clover package coverage for Collections, respectively.



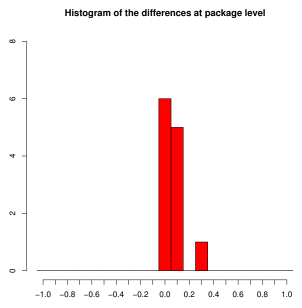


Figure 12. Scatter of static and Clover coverage and histogram of the coverage differences for Collections at package level.

$70-80\%$ and in the $80-90\%$ ranges, where a lower and higher number of classes was recognized, respectively.

Figure 10 depicts a scatter plot for estimate and true value (with a diagonal line where the estimate is correct), and a histogram of the differences between estimated and true values. For a large number of classes estimated coverage matched true coverage. This can be observed in the scatter plot, in which several points are on the diagonal, and in the histogram, by the large bar above $0$. In the histogram, we can additionally see that, on both sides of the bar differences decrease indicating a small estimation error.

Recalling Spearmans's correlation value, $0.549$, we understand that the correlation is not higher due to the considerable number of classes for which static coverage overestimates or underestimates results without a clear trend.

Package-level results are shown in Figure 11. In contrast to class-level results, the histograms at package level do not look so similar. In the $50-70\%$ range static estimate failed to recognize coverage. Comparing to Clover, in the $70-80\%$ and $90-100\%$ ranges static estimate reports lower coverage and in the $80-90\%$ range reports higher coverage.

Figure 12 show the scatter plot and the histogram of differences at the package level. In the scatter plot we can observe that for a significant number of packages, static coverage was overestimated. However, in the histogram, we see that for $6$ packages estimates are correct, while for the remaining $5$ packages, estimates are slightly overestimated. This is in line with the correlation value of $0.776$, from Table III.

Thus, for the Collections project, the estimated coverage at class and package level can be considered good.

### C. *Evaluation*

Static estimation of coverage is highly correlated with true coverage at all levels for a large number of projects. The results at system level allow us to positively answer the question: can test coverage be determined without running tests? The tradeoff, as we have shown, is some precision loss with a mean of the absolute differences around $9\%$. The analysis on $52$ releases of the Utils project gave us additional confidence in the results for system coverage. We observed, that static coverage not only can be used as predictor for real coverage, but it also detects coverage fluctuations.

According to our expectations, static coverage at package level reported better correlation than at class level. However, the correlation for all systems at package level is just slightly higher than for class level. We expected that grouping classes would cancel more imprecision and hence provide better results. However, this is not always the case. For a small number of packages, static coverage produces large overestimations or underestimations, causing outliers, having a negative impact in both correlation and dispersion.

At class level, the correlation values are quite high, but the dispersion of differences is still rather high to consider static analysis a good estimator for class-level coverage.

We have observed that control flow and dynamic dispatch cause an overestimation, while frameworks/libraries cause underestimation of coverage which, in some cases, is responsible for coverage values lower than true coverage.

## V. Related Work

We are not aware of any attempt to compute test coverage using static analysis. However, there is a long record of work with which we share underlying techniques.

Koster et al. [11] introduced a new test adequacy criteria, *state coverage*. A program is state-covered if all statements that contribute to the output or side effects are covered by a test. The granularity of state coverage is at the statement level, while our technique is at the method level. State coverage limits coverage to system only, while we report system, package and class coverage. State coverage also uses static analysis and slicing. However, while a data flow graph from bytecode is used, our technique uses a call graph extracted from source code. Koster et al. do not identify sources of imprecision, and uses as case study a small open-source project. We described sources of imprecision and presented a comparison using several projects both proprietary and open-source from small to large sizes.

Ren et al. [12] proposes a tool, Chianti, for change impact analysis. The tool analyses two versions of a program, original and modified. A first algorithm identifies tests potentially affected by changes, and a second algorithm detects a subset of the code potentially changing the behavior of tests. Both algorithms use slicing on a call graph annotated with change information. Ren et al. use dynamic analysis for deriving the graph, however in a previous publication of Chianti [13], Ryder et al. used static analysis. Our technique also makes use of graph slicing at method level granularity with the purpose of making the analysis amenable. Chianti performs slicing twice. First, from production code to test code and second from tests to production code. By contrast, our technique only performs slicing once, from tests to production code, to identify production code reached by tests. Finally, despite using a similar technique to Chianti our purpose is test coverage estimation.

Binkley [14] proposes a regression test selection (RTS) technique to reduce both the program to be tested and the tests to be executed. This technique is based on two algorithms. The first algorithm extracts a smaller program, *differences*, from the semantics differences between a previously tested program, *certified*, and the program to be tested, *modified*. The second algorithm identifies and discards the tests for which *certified* and *differences* produce the same result, avoiding the execution of unnecessary tests. Both algorithms make use of static slicing (backward and forward) over a system dependence graph, containing statement-level and control-flow information. By contrast, our approach involves forward slicing over a call graph, which contains less information and requires a simpler program analysis to construct.

Rothermel and Harrold [15], present comprehensive surveys in which they analyze and compare thirteen techniques for RTS. As previously stated, RTS techniques attempt to reduce the number of tests to execute, by selecting only those who cover the components affected in the evolution process. RTS techniques share two important ingredients with our technique: static analysis and slicing. However, while most RTS techniques use graphs with detailed information, e.g., system dependence graphs, program dependence graphs, data flow graphs, our technique uses less detailed information. Moreover, we also share with these techniques the basic principles. RTS techniques analyze code that is covered by tests in order to select the tests to run. Our technique, on the other hand, analyzes code under test in order to estimate coverage.

Harrold [16] and Bertolino [17], present a survey about software testing and research challenges to be met. Testing is a challenging and expensive activity and there are ample opportunities for improvement in, for instance, test adequacy, regression test selection and prioritization. We have shown that static coverage can be used to assess test adequacy. Rothermel et al. [18] surveys nine test prioritization techniques from which four are based on test coverage. These four techniques assume the existence of coverage information produced by prior execution of test cases. Our technique could be used as input replacing the execution of tests. Finally, Lyu [19] has surveyed the state of the art of software reliability engineering. Lyu describes and compares eight reports of the relation between static coverage and reliability. In the presence of a very large test suite, our technique could be used to substitute the coverage value by an approximation and used as input of a reliability model.

## VI. Concluding remarks

We have described an approach for estimating code coverage through static analysis. The approach does not require detailed control or data flow analysis in order to scale to very large systems and it can be applied to incomplete source code. We have discussed the sources of imprecision of the analysis and we have experimentally investigated its accuracy. The experiments have shown that at the level of complete systems, the static estimate is strongly correlated with the true coverage values which, in the context of software quality assessment, proves to be satisfactory. At the level of individual packages or classes, the difference between the estimated and true values is small in the majority of cases revealing opportunities for further improvement.

*Future work:* Several avenues of future work are worth exploration. The accuracy of results could be improved by refining the static derivation of call graphs: commonly used frameworks can be factored into the analysis, bytecode analysis can be used to recognize library/framework calls, and methods called by reflection can be estimated.

Estimation of statement coverage rather than method coverage could be attempted. To not rely on detailed statement-level information, a simple LOC count for each methods could be used to estimate statement coverage.

It is also of interest to investigate the use of a more detailed dependency analysis. The use of a system dependence graph and intra-procedural analysis could, as in Binkley [14] reduce the imprecision caused by dynamic dispatch and improve our estimation and allow us to do more fine-grained analysis enabling the estimation of statement or branch coverage. Additionally, we would like to adopt techniques to predict the execution of particular code blocks, such as estimation of the likelihood of execution by Boogerd et al. [20]. Although sophistication in the static analysis may improve accuracy the penalties, e.g., scalability loss, should be carefully considered.

We are also interested in combining our technique with other work, namely from Kanstrén [21]. Kanstrén defines a test adequacy criteria in which code is considered as test covered if it is tested at both the unit level and the integration level. These levels are measured using the distance from the test to the method via the call chain. The call chain is derived dynamically by executing tests. Our plan is to replace the dynamic analysis by our static analysis and quantitatively and qualitatively measure test quality. We would like to experiment replacing the dynamic analysis to determine the levels by our static analysis and quantitatively and qualitatively measure test quality.

Finally, we are currently extending our approach to C#.

*Acknowledgements:* Thanks to Ilja Heitlager for inspiring discussions, and to David Binkley, Martin Bravenboer, Eric Bouwers, Alexandre Bergel, Mário Pinhal, Miguel Ferreira and José Pedro Correia for providing helpful feedback about our work. Finally, we would like to thank Semmle Ltd., for the support with SemmleCode. The first author is supported by Fundação para a Ciência e a Tecnologia, Portugal, under grant number SFRH/BD/30215/2006.

## References

[1] K. Beck, "Simple smalltalk testing: with patterns," *Smalltalk Report*, vol. 4, no. 2, pp. 16–18, 1994.

[2] I. Heitlager, T. Kuipers, and J. Visser, "Observing unit test maturity in the wild," 13th Dutch Testing Day 2007.

[3] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *AST'06*. ACM, 2006, pp. 99–103.

[4] A. van Deursen and T. Kuipers, "Source-based software risk assessment," in *Proc. ICSM*, 2003, p. 385.

[5] T. Kuipers, J. Visser, and G. de Vries, "Monitoring the quality of outsourced software," in *TOMAG'07*, J. van Hillegersberg *et al.*, Eds., 2007.

[6] A. Lakhotia, "Graph theoretic foundations of program slicing and integration," University of Southwestern Louisiana, Tech. Rep. CACS TR-91-5-5, 1991.

[7] O. de Moor *et al.*, ".QL: Object-oriented queries made easy," in *GTTSE'07*, ser. LNCS, R. Lämmel and J. Visser, Eds. Springer, 2008, to appear.

[8] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[9] C. Spearman, "The proof and measurement of association between two things," *Amer. J. of Psychology*, vol. 100, no. 3/4, 1987.

[10] T. W. Anderson and D. A. Darling, "Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes," *Ann. Math. Statist.*, vol. 23, no. 2, pp. 193–212, June 1952. [Online]. Available: http://projecteuclid.org/euclid.aoms/1177729437

[11] K. Koster and D. Kao, "State coverage: a structural test adequacy criterion for behavior checking," in *Proc. ESEC/FSE*. ACM, 2007, pp. 541–544.

[12] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *OOPSLA*. ACM, 2004, pp. 432–448.

[13] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *PASTE'01*. ACM, 2001, pp. 46–53.

[14] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Softw. Eng.*, vol. 23, no. 8, pp. 498–516, 1997.

[15] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, Aug 1996.

[16] M. J. Harrold, "Testing: a roadmap," in *ICSE'00*. New York, NY, USA: ACM, 2000, pp. 61–72.

[17] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *FOSE'07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103.

[18] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, Oct 2001.

[19] M. R. Lyu, "Software reliability engineering: A roadmap," in *FOSE'07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 153–170.

[20] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *SCAM'06*, 2006, pp. 149–160.

[21] T. Kanstrén, "Towards a deeper understanding of test coverage," *J. Softw. Maint. Evol.*, vol. 20, no. 1, pp. 59–76, 2008.