Static Estimation of Test Coverage

Tiago L. Alves
Universidade do Minho, Portugal, and
Software Improvement Group, The Netherlands
tiagomlalves@gmail.com

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

Abstract

Test coverage is an important indicator for unit test quality. Test coverage can be computed by tools such as Clover or Emma by first instrumenting the code with logging functionality, and then running the instrumented code to log which parts are executed during unit test runs. Since computation of test coverage is a dynamic analysis, it presupposes a working installation of the analyzed 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 software libraries or hardware platform. The installation procedure may not be automated or even documented. Instrumentation may not be feasible, e.g. due to space or time limitations in case of embedded software.

In this paper, we propose a method for estimating test coverage through static analysis only. The method uses slicing of static call graphs to estimate the actual dynamic test coverage. We explain the method and its implementation in an analysis tool. We validate the results of the static estimation by comparison to actual values obtained through dynamic analysis using Clover.

1. 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 [5].

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 their 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 [12]. These tools work by instrumenting the code with logging functionality before execution. The logging information collected during execution is then aggregated and reported to the developer. For example, the Clover[2] tool instruments Java source code, and reports statement coverage and branch coverage at the level of methods, classes, packages, and overall system. The Emma[3] tool instruments Java byte code, and reports statement coverage and method coverage at the same levels. The detailed reports of such tools provide highly valuable input for developers to increase or maintain the quality of their test code.

Computing code coverage involves running the application code and therefore requires a working installation of the software being analyzed. 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 [11, 8, 9, 3], 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.

The instrumentation of source or byte code by coverage tools is another complicating factor. In the context of embedded software, for instance, after instrumentation applications may simply not run or may display essentially altered behavior due to space or performance changes.

The question then naturally arises whether code coverage by tests could possibly be determined without actually

[1]sunit.sourceforge.net, www.junit.org, www.nunit.org

[2]http://www.atlassian.com/software/clover/

[3]http://emma.sourceforge.net/

running them? And what trade-off can be made between sophistication of such a static analysis and its accuracy?

In this paper we begin to answer these questions. We propose a static analysis for estimating code coverage, based on slicing of call graphs (Section 2). We discuss the sources of imprecision inherent in this analysis as well as some possible countermeasures (Section 3). We experimentally assess the quality of the estimate by comparing static estimates to dynamically determined code coverage results for a range of proprietary and open source software systems (Section 4). We discuss related work in Section 5 and we conclude with a summary of contributions and possible avenues of future work in Section 6.

2. Approach

Our approach for statically estimating code coverage involves reachability analysis (slicing) on a graph structure obtained from source code. The graph contains information about method invocations and code structure. The granularity of the graph is at the method level, and detailed control or data flow information is not needed. Test coverage is estimated by calculating the ratio between the production code reached from test code and the overall production code.

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, including both production and test code, a graph G is extracted which records both structural information and call information. Also, the test classes are identified and collected in a set.

2. Starting from test classes, the graph is sliced, primarily along call edges, to collect all methods covered by these test classes.

3. For each non-test 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.

4. The final estimates at class, package, and system level are obtained as ratios from the counts per class.

The proposed approach is designed with a number of desirable characteristics in mind:

- Only static analysis is needed. The graph contains call information extracted from source code.

- Scales to large systems. Granularity is limited to the method level to keep whole-system analysis tractable.

- Robust against partial availability of source code. Missing information is not blocking, though it may lead to less accurate estimates.

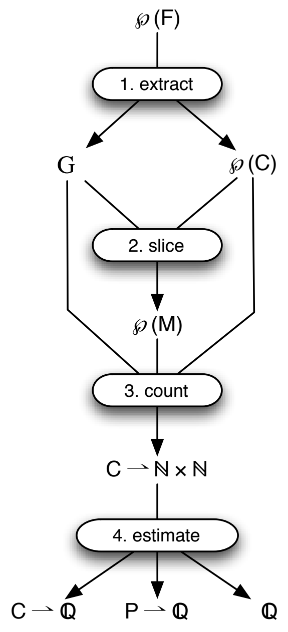


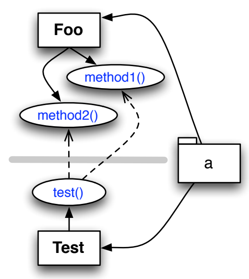
Figure 1. Overview of the approach. The input is a set of files (F). From these files, a call graph is constructed (G). Also, the test classes of the system are identified (C). Slicing is performed on the graph with the identified test classes as entry points, and the production code methods (M) in the 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 level, package level, and system level.

The extent to which these desirable properties are actually realized will become clear in Section 4. First, we will explain the various steps of our approach in more detail.

2.1. Graph construction

Through static analysis of source code, a graph structure is derived in which packages, classes, interfaces, and methods are represented, as well as various relations between them. An example is provided in Figure 2. We will explain the node and edge types that are present in such graphs.

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, classes, interfaces, and methods. Thus, the set V of vertices can be partitioned into four subsets which we will write as $N_n \in V$ where node type



```
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). Production code is depicted above the line and test code below.

$n \in \{P, C, I, M\}$. In the various figures in this paper, we will represent packages as folder icons, classes and interfaces as rectangles, and methods as ellipses.

The sets of nodes that represent classes, interfaces, and methods can also be partitioned to differentiate between test and production code. We write N_n^c where code type $c \in \{P, T\}$. 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 information and call information. For structural information two types of edges are used:

- The *defines type* edges (DT) connect a packages to its classes and interfaces.
- The *defines method* edges (DM) connect classes and interfaces to their methods.

For call information, three types of edges are used:

- A *call* edge (C) represent the invocation of a method by its caller, which is usually another method. In case of method invocations in initializers, the origin of a call edge is a class or interface. In any case, 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 *overloading call* edge (OC) is constructed between a caller and any overloaded method definition to which it might be resolved at runtime.

The set of edges is actually a relation between vertices such that $E \subseteq \{(u, v)_e \mid u, v \in V\}$, where $e \in \{DT, DM, C, VC, OC\}$.

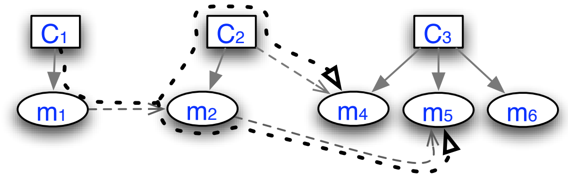


Figure 3. Modified graph slicing algorithm in which calls are taken into account originating from both methods and class initializers. The grey arrows are edges in the graph and the black arrow depicts the slicing traversal.

We write E_e for the five 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 three types of call edges will be provided in Section 3.1 where we discuss sources of imprecision for our static coverage estimation.

2.2. Identifying test classes

Production and test code are usually stored in different file system paths and/or made easily distinguishable by name. But such conventions are not always strictly adhered to. Furthermore, testing utility code is often not consistently categorized under either production or test code.

For these reasons, a slightly finer approach is needed to identify test code correctly. As a drawback, this approach must be tailored for each testing framework. For example, to recognize JUnit framework test classes, we check for each class whether it extends `TestCase` or `TestSuite`, or we check for the presence of test annotations.

2.3. Slicing to collect covered methods

In the second step of static coverage estimation approach, graph slicing [10] is applied to collect all methods covered by a given set of test classes. Basically, we use the identified set of test classes as slicing criteria (starting points). We then follow *define method* edges and the various kinds of *call* methods in forward direction to reach all covered methods. In addition, we modified the graph slicing algorithm to take into account call edges originating in class 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. The modified slicing algorithm is depicted in Figure 3.

The modified slicing algorithm can be defined as follows. We write $n \xrightarrow{call} m$ for an edge in the graph that represents that a node n calls a method m, where the call type can be vanilla, virtual, or overloaded. 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. 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 node 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 in a straightforward way. Alternatively, a relational querying language such as .QL [4] can be used to express these definitions almost directly.

2.4. Count methods per class

The third step in our algorithm is to compute the two fundamental metrics for our static test coverage estimation:

Number of defined methods per class (DM) defined as $DM : n_C \to \mathbb{N}$. This metric is calculated by counting the number of outgoing *define method* edges per class.

Number of covered methods per class (CM) defined as $CM : n_C \to \mathbb{N}$. 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 computed in the previous step.

These computed static metrics are then stored in a finite map $n_C \rightharpoonup \mathbb{N} \times \mathbb{N}$ for further processing.

2.5. Estimate static test coverage

After computing the two basic metrics we can obtain derived metrics: coverage per class, packages, and system.

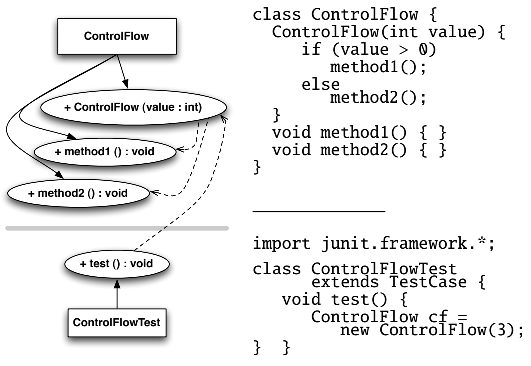
Class coverage Class coverage (more precisely: method coverage at the class level) is simply computed by the ratio between covered methods and defined methods per class:

$$CC(c) = \frac{CM(c)}{DM(c)} \times 100\%$$

Package coverage Method coverage at the package level is the ratio between the total number of covered methods and the total number of defined methods per package:

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

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



```
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.

System coverage system coverage is computed by the ratio between the total number of covered methods and the total number of defined methods in the overall system:

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

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

3. Imprecision

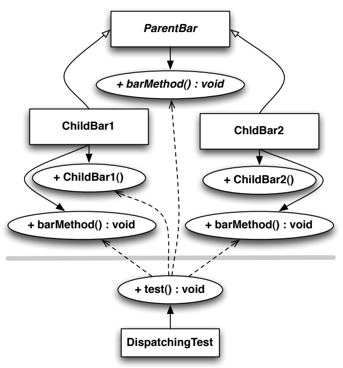
Our coverage analysis is based on a statically derived call graph, in which the structural information is exact and the method call information is an approximation of the dynamic execution. In this section we discuss the various sources of imprecision, such as control flow, dynamic dispatching, overloading, library/framework calls, and test classes recognition. We also discuss how to deal with the imprecision.

3.1. 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, `method1` is called if the `value` variable is greater than zero. Otherwise `method2` is called. In the test, we can observe that 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. Thus, our static call graph will unconditionally contain the call edge.

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

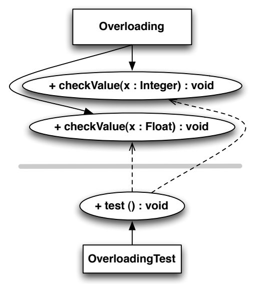
Dynamic dispatching Figure 5 presents an example where the graph has imprecision due to dynamic dispatching. A parent class `ParentBar` defines a `barMethod`



```
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.


Overloading
+ checkValue(x : Integer) : void
+ checkValue(x : Float) : void
+ test () : void
OverloadingTest

```
class Overloading {
  void checkValue(x:Integer) {}
  void checkValue(x:Float) {}
}
```

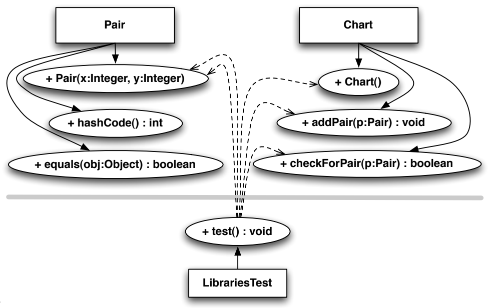
```
import junit.framework.*;
class ControlFlowTest
      extends TestCase {
   void test() {
      Overloading o =
         new Overloading();
      o.checkValue(3);
}  }
```

Figure 6. Imprecision: method overloading.

method, 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. Thus, during test execution, the `barMethod` of `ChildBar1` will be called. Our static analysis, however will identify all three implementations of `barMethod` as potential call targets.

Overloading Figure 6 presents an example where the graph has imprecision due to overloading. The class `Overloading` contains two methods with the same name but with different argument types: Integer and Float. In the test we call the `checkValue` method with the constant value 3 and the method with Integer argument is called. However, our call graph is constructed without dynamic type analysis and will include calls to both methods.

Frameworks / Libraries Figure 7 presents an example where the graph has imprecision caused by calls to a library or framework 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


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 7. Imprecision: library calls.

coordinate $(3, 5)$ is added to the chart. Then another object with the same coordinates is created and checked to exist in the chart. The source code does not contain calls to the `hashCode` or `equals` methods. Still, dynamically, when a `Pair` object is added to the set, the `hashCode` method is called. Moreover, when checking whether an object exists in a set, both methods `hashCode` and `equals` are called. These calls are not present in our call graph.

Identification of test code As previously stated, the recognition of test code is done by analyzing classes for JUnit library references. A class is considered test code if it uses the JUnit library.

However not all test code needs JUnit. In particular test helper classes that containing common functionality for testing, e.g. common initializations, do not require JUnit and such they will be considered as production code, causing yet another source of imprecision.

3.2. Dealing with imprecision

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, we have decided to opt for the optimistic approach. A pessimistic approach would lead to a large number of false negatives (methods that are erroneously reported not to be covered). In the optimistic approach, methods reported not be covered are with high certainty indeed not covered.

The downside of the optimistic approach is that we can not be consistently optimistic for all the uncertainties previously mentioned: imprecision caused by library and framework calls is always pessimistic. In the sequel, we will investigate experimentally what the consequences of these choices are for the accuracy of the analysis.

4. Experimentation

We experimentally compared the results of static estimation of coverage against dynamically computed coverage.

4.1. Experimental design

Systems analyzed For this experiment, we analyzed Java systems fulfilling two criteria:

- Use of JUnit as unit testing framework.
- Having an already available clover report either in XML or HTML format.

The second criterion prevented the need for configuration, installation, and execution of the systems.

The systems are listed in Table 1. As can be observed, the list is diverse both in terms of size (ranging from small to medium, $6k$ – $80k$ lines of code), and in scope: Pacman is a very small system developed for educational purposes, Dom4j is an open-source library and PMD, Architect and DepFinder are open-source tools[4], G System and R System are proprietary systems and Utils is a proprietary library.

Measurement For the dynamic coverage measurement, we used data produced by the Clover tool. We did not run Clover ourselves but relied on Clover reports available for the various systems, either in XML or HTML format. When available in XML format, XLST transformations were used to extract the required information and convert it to a flat textual format. When only available as HTML files, we applied Unix scripts to extract the same 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 [4]. The results were exported to a textual format.

The textual files for static and clover coverage, for both package and class, were then further processed and combined into a spreadsheet with a custom built Java tool. In essence, the spreadsheet contains tables with the information from the files, and comparison tables for class and package. The spreadsheets were used as input for statistical processing and charting.

[4]`www.dom4j.org`, `pmd.sourceforge.net`, `www.sqlpower.ca/page/architect`, `depfind.sourceforge.net`

System name	Static	Clover	Differences
Pacman	84.53%	90.61%	−6.08%
G System	88.37%	94.81%	−6.44%
Dom4j	60.69%	41.00%	12.27%
Utils	72.95%	68.73%	4.22%
PMD	77.77%	65.50%	12.27%
Architect	48.98%	35.30%	13.68%
DepFinder	62.14%	70.08%	−7.94%
R System	64.54%	72.46%	7.92%

Table 2. Static and clover coverage, and coverage differences at system level.

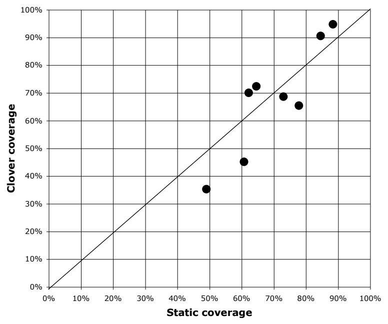


Figure 8. Scatter plot comparing static and dynamic coverage for all systems.

Statistical analysis For statistical analysis we used SPSS version 11 for Mac. We created histograms to inspect the distribution of true coverage and static estimate. 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 various basic statistics, such as median and interquartile range. Since the data does not follow normal distributions, we applied Spearman, a non-parametric algorithm to investigate the correlation of true and estimated coverage.

4.2. Experiment results

We discuss the results of the experiment, first at the level of complete systems. Then we look at the class and package level results and we look at two systems in more detail.

Table 2 and the scatter chart in Figure 8 show the true and estimated system-level coverage for all systems analyzed. As can be observed in Table 2, the differences range

System name	Version	Author / Owner	Description	LOC	#Packages	#Classes	#Methods
Pacman	3.0.3	Arie van Deursen	Game used for OOP educational purposes	2987	2	20	181
G System	14/02/08	C Company	System for managing leasing contracts	6265	15	53	385
Utils	20/02/08	SIG	Library of tools for static code analysis	23604	35	260	2549
Dom4j	1.6.1	Dom4j	Library for working with XML	42863	14	143	2339
PMD	3.8	PMD	Java static analysis checker tool	51219	40	455	3475
Architect	0.9.9	SQLPower	Data modeling and profiling tool	58477	17	220	2056
DepFinder	1.2	DepFinder	Java bytecode analysis tool	73861	12	260	2920
R System	14/02/08	C Company	G System database synchronization tool	79776	62	600	5578

Table 1. Characterization of the systems used in the experiment order by LOC.

System	Spearman		Median		Interquartile Range	
name	Class	Package	Class	Package	Class	Package
Pacman	0.275	1	0	-0.086	0.093	-
G System	0.777^{**}	0.694^{**}	0	0	0	0.046
Utils	0.737^{**}	0.825^{**}	0	0.01	0.038	0.109
Dom4j	0.557^{**}	0.625^{*}	0.17	0.099	0.373	0.243
PMD	0.702^{**}	0.693^{**}	0	0.066	0.128	0.189
Architect	0.504^{**}	0.5^{*}	0	0.064	0.28	0.197
DepFinder	0.659^{**}	0.396	0	-0.004	0.13	0.132
R System	0.752^{**}	0.652^{**}	0	-0.1	0.01	0.186

Table 3. Statistical analysis reporting correlation between static and clover coverage, and median and interquartile ranges for coverage differences at system level.

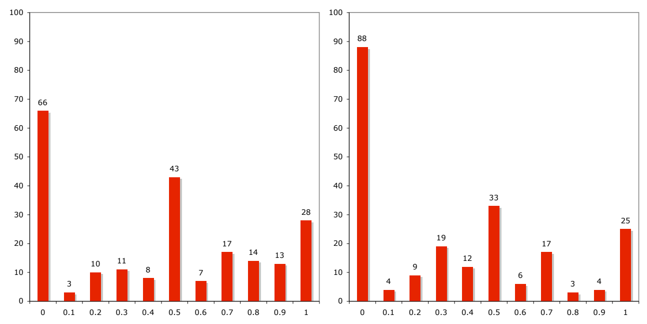


Figure 9. Histograms of static and clover class coverage for the Architect system.

from 4 to 14 percent points. On average, the absolute value of the differences is 9.25 percent points. For half of the systems the static coverage is pessimistic (negative difference) while for the other half is optimistic (positive difference). The chart in Figure 8 shows that static coverage values are very close to the diagonal, which depicts the true coverage value. Moreover, Spearman correlation reports a value of 0.976 with high significance, meaning that true and estimated system-level coverage are highly correlated.

Despite these encouraging system-level results, it is also important, and interesting, to analyze the results at package and class levels. Table 3 reports for each system the following indicators for correlation, central tendency, and dispersion of the class-level and package-level results: the Spearman correlation coefficient and significance, median and interquartile range (IQR).

At class level, all systems except Pacman, report high correlation with very high significance. The low Spearman correlation value for Pacman is probably related to its small size. With respect to the median, all systems are centered in zero, except Dom4j which is slight optimistic. The IQR, on the hand, varies ranging from extremely low values (G System) to relatively high values (Dom4j and Architect) meaning that the dispersion is not uniform among systems. Also, for the systems with a high IQR Spearman reported lower correlation values.

At package level, all systems except DepFinder and again Pacman report correlation values with high or very high significance. In the case of Pacman, correlation factor is high, but without significance due to the sample size of only 2 packages. For DepFinder, the low correlation is explainable by the low number of packages and by the existence of two outliers. Would these be removed, then correlation jumps to 0.766 with high significance. Regarding the median, we can observe that only G System reports a value of 0 while all the others report values close to zero. The IQR show more homogeneous values then at class level, with values around 18% except for Pacman (which does not have IQR due to its sample size of 2) and G System.

We now discuss the results for two individual systems: Architect with relatively low coverage and the R System with relatively high coverage.

Architect In Figure 9 the histograms depict the distributions of estimated and of true class-level coverage for the Architect system. There are three noticeable peaks, at 0%, 50% and 100%. These correspond to classes without any coverage, classes that are half-tested, and classes with full coverage. Comparing the two histograms we can observe that static coverage was fairly accurate to estimate clover coverage in a big percentage of the coverage range and successfully recognized the same peaks as in clover coverage. Exceptions, besides slight variations, are at 0%, 50% and in the range 80-90% coverage where the estimate is

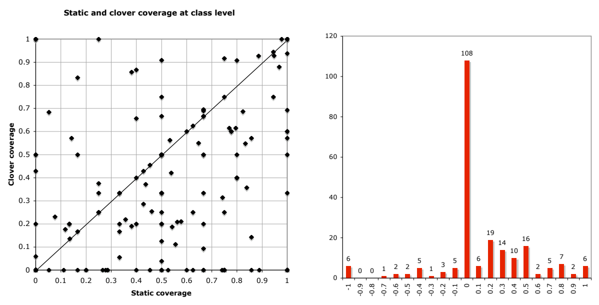


Figure 10. Scatter of static and clover coverage and histogram of the coverage differences for Architect at class level.

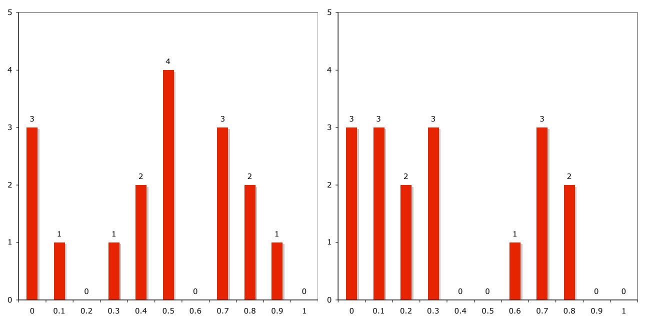


Figure 11. Histograms of static and clover package coverage for the Architect system.

clearly pessimistic compared to the true coverage, and at 30% where the estimate is slightly optimistic.

Figure 10 depicts a scatter chart 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. In the scatter chart we can observe that several points are on the diagonal, a smaller number of points lies above the line (pessimistic) and a larger amount of points lies below the line (optimistic). The histogram shows that there is a big number of classes for which static coverage matched clover coverage (difference near to zero).

Recalling the value for spearman correlation (0.504 with very high significance) we can now understand why the correlation value is not higher: there is a significant number of classes whose static coverage reported optimistic results, as can be seen on the right side of the histogram.

Package-level results are shown in Figure 11. In contrast to what was observed for class coverage, the histograms at package level do not look so similar. The first noticeable thing is that the static estimate reported excess coverage in the range 40% – 50% and at 90% coverage. The estimate is accurate at 0% and in the range 70% – 80% coverage and

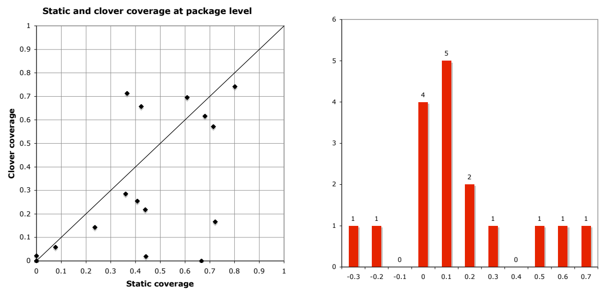


Figure 12. Scatter of static and clover coverage and histogram of the coverage differences for Architect at package level.

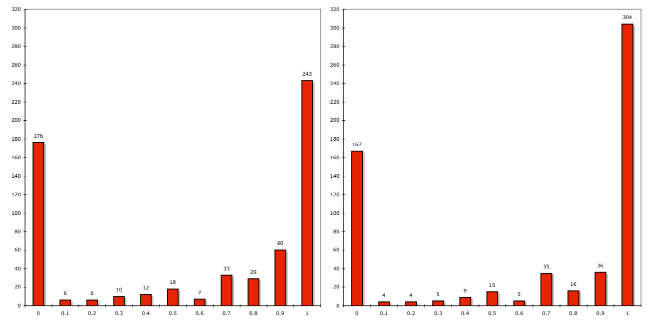


Figure 13. Histograms of static and clover class coverage for the R system.

elsewhere slightly pessimistic.

Figure 12 show the scatter chart and the histogram of differences at the package level. In the scatter chart we can observe that we have just 4 packages for which static coverage was pessimistic, while a larger number of packages has optimistic values. In the histogram, we see that for a total of 17 packages 4 estimates correct (close to 0), while another 5 are slightly optimistic (also observed in the scatter plot).

Thus, for the Architect system, the estimated coverage values at class and package level can be considered fair, but not excellent. This is in line with the correlation coefficient of about 0.5 reported in Table 3.

R System analysis Figure 13 shows histograms of estimated and true class-level coverage for the R System. The charts are similar and show two peaks, at 0% and 100%, a slight raise between two peaks. Thus, there are two extremes corresponding to classes that are not covered and classes that are fully covered. In the range in between, there is a smaller number of classes that grows towards 100% of coverage. From these histograms, we can also notice that at 0% of coverage, the static estimate is slightly pessimistic

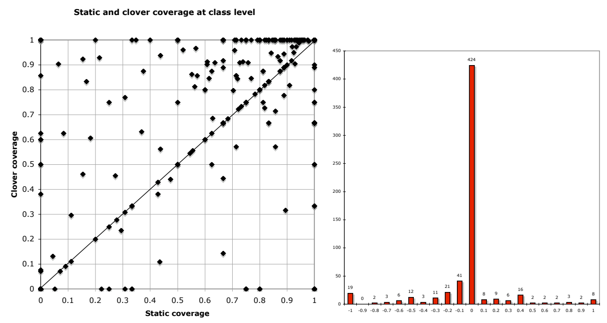


Figure 14. Scatter chart comparing static and clover class coverage and histogram of the coverage differences for the R system.

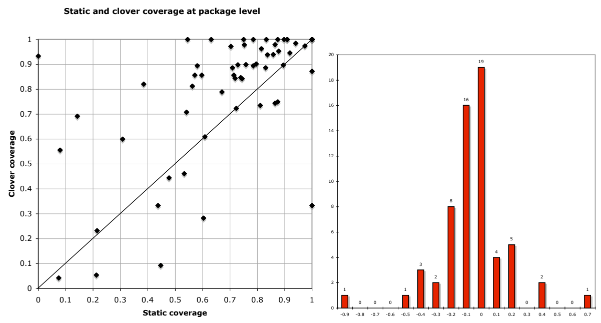


Figure 16. Scatter chart comparing static and clover package coverage and histogram of the coverage differences for the R system.

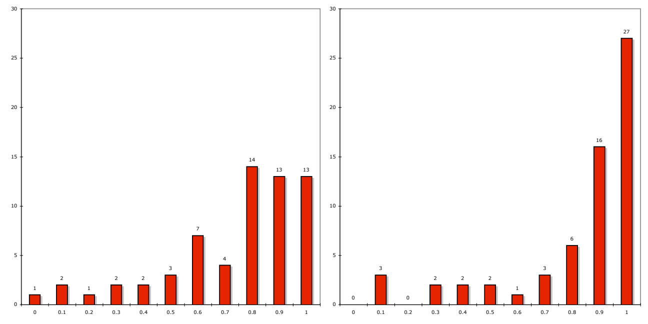


Figure 15. Histograms of static and clover package coverage for the R system.

while at 100% it is more optimistic.

Figure 14 depicts again a scatter plot of estimated against true values, and a histogram of their differences. As it can be observed there are points all along the line that defines the perfect match between static and clover coverage. On average, estimation is pessimistic, with a higher number of points above the line than below the line.

Another interesting characteristic present in the scatter plot is the cluster of classes that can be found in the top right corner of the chart above the line. If we just look to the scatter chart this might lead to the erroneous interpretation that there is a large group of classes for which estimation is pessimistic. However, if we look to the histogram, we can see that the number of classes with pessimistic estimates is much smaller than the number of classes with accurate estimates. Finally, is interesting to notice that this histogram is very similar to the one in Figure 10 with a peak at zero and residual values around that decrease toward the extremes.

The package-level results of the R System are shown in Figure 15. As for the Architect system, the histograms are not very similar even though we have more data points than in Architect. Figure 16 shows a scatter plot and a histogram of differences. Interestingly, the scatter plot for package resembles the scatter chart for class level with respect to the cluster of data points observed above the line for coverages higher than 70%. Moreover, we observe that the static estimate is pessimistic for coverages above that value.

4.3. Evaluation

Static estimation of coverage is highly correlated with true coverage at all levels, with only a few exceptions. The results at system level allow us to positively answer the question: can test coverage be measured without running tests? The tradeoff, as we have shown, is some lack of precision which in average is around 9%.

Against our expectations, static coverage at package level reported worse correlation than at class level. Although at beginning this seemed counter-intuitive since we were expecting that grouping classes would cancel imprecision and hence provide better results, we discovered why this was not always the case. For several systems, we observed that for a small number of packages static coverage reports values that are either optimistic or pessimistic causing big outliers. As consequence, these outliers create a negative impact both in 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.

5. Related Work

We are not aware of any attempt reported in the literature to compute test coverage using static analysis. However, some of the techniques we employ, such as static call graphs, and slicing from test code to production code (and vice versa), have been applied before for other purposes such as change impact analysis or regression test selection.

Koster et al [7] introduced a test adequacy criteria called *state coverage*. A program is state-covered if all statements that contribute to the output or side effects are covered by a test. They also use a statically extracted graph and slicing to estimate coverage. However, while the authors estimate state coverage we report code coverage at system, package and class levels. Also, they use a static data flow graph extracted from byte code, while we use a static call graph extracted from source code. They do no identify sources of imprecision as we do, and they only measure correlation for a small open-source project while we made a comparison using several projects both proprietary and open-source from small to medium sizes.

6. 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. It scales well to very large systems and it can be applied to incomplete source code. We have discussed the sources of imprecision inherent in 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 and, in the context of software quality assessment, these results are considered very good. At the level of individual classes, the difference between the true value and its estimate is small in the majority of cases.

6.1. Future work

Several avenues of future work are worth exploration. Modifications to the static derivation of call graphs can be made to improve accuracy of the results. For example, commonly used frameworks can be factored into the analysis to fill in library/framework calls that are currently omitted.

The penalties and benefits should be investigated of resorting to techniques for data and control flow analysis and for partial evaluation. Additional sophistication in the static analysis may improve accuracy, but may reduce scalability. For example, techniques for estimating the likelihood of execution of particular code blocks could be adopted [2], but then a more detailed graph structure would be required.

The approach could be modified to estimate statement coverage rather than method coverage. For this purpose, the LOC for all methods could be taken into account.

We are currently extending our approach to other languages with C# as next target. Also, we are customizing the analysis to identify test code for further testing frameworks.

Finally, we are interested in combining this technique with other works, namely [6], to gain test quality insight.

Acknowledgements Thanks to Ilja Heitlager for inspiring discussions. 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*, 4(2):16–18, 1994.

[2] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *SCAM '06: Proc. 6th Int. Workshop on Source Code Analysis and Manipulation*, pages 149–160. IEEE Computer Society, 2006.

[3] E. Bouwers and R. Vis. Multidimensional software monitoring applied to erp. In C. Makris and J. Visser, editors, *Proc. 2nd Int. Workshop on Software Quality and Maintainability*, ENTCS. Elsevier, 2008. To appear.

[4] O. de Moor et al. .QL: Object-oriented queries made easy. In R. Lämmel and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering II*, LNCS. Springer, 2008. To appear.

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

[6] T. Kanstrén. Towards a deeper understanding of test coverage. *J. Softw. Maint. Evol.*, 20(1):59–76, 2008.

[7] K. Koster and D. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *Proc. 6th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 541–544. ACM, 2007.

[8] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In M. Piattini and M. Serrano, editors, *Proc. 1st Int. Workshop on Software Audit and Metrics, (SAM 2004)*, pages 118–128. INSTICC Press, 2004.

[9] T. Kuipers, J. Visser, and G. de Vries. Monitoring the quality of outsourced software. In J. van Hillegersberg et al., editors, *Proc. Int. Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007)*. Center for Telematics and Information Technology (CTIT), The Netherlands, 2007.

[10] A. Lakhotia. Graph theoretic foundations of program slicing and integration. Technical Report Technical Report CACS TR-91-5-5, The Center for Advanced Computer Studies, University of Southwestern Louisiana, 1991.

[11] A. van Deursen and T. Kuipers. Source-based software risk assessment. In *ICSM '03: Proc. Int. Conference on Software Maintenance*, page 385. IEEE Computer Society, 2003.

[12] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *AST '06: Proc. 2006 int. workshop on Automation of Software Test*, pages 99–103. ACM, 2006.