Quality Metrics: Test Coverage Analysis for Smalltalk

Mark Connell
Open Environment Australia*
Mark.Connell@c031.aone.net.au

Tim Menzies
University NSW†
timm@cse.unsw.edu.au

Abstract

The growth in the use of object technology has led to a corresponding requirement for improved quality in object-oriented (O-O) software. If the holy grail of software re-use is to be achievable then there has to be a large degree of confidence in the component building blocks. With the possibility of a single O-O component being used in many software systems the need for greater confidence in the quality of the component increases. Quality assurance comes from testing and the use of metrics to give some quantitative measure of quality. The requirement for research, leading to the availability of methodologies, systems and tools, to assist with testing and quality assurance is growing. This paper deals with the use of coverage analysis in testing in Smalltalk. Various methods are described and discussed. Bytecode interpreter approaches are rejected due to the non-standard nature of bytecodes in different versions of Smalltalk. We endorse a textual insertion method due to its simplicity and its applicability over different Smalltalk implementations. However, we caution that this textual insertion method has fixed limits.

1 Introduction

In [Whitty 96a] Whitty details current research work in the area of O-O metrics. The data in this report shows the increase of work in this area, corresponding with the increased use of object technology and the increased requirement and use of metrics by industry. However, a closer analysis of the data indicates a shortage of research related to the testing of Smalltalk systems. The availability of tools and systems has not kept pace with the increased use of Smalltalk in industry. The data in [Whitty 96b] shows that out of a total of 269 articles only 15 entries, or 6%, are concerned with the field of software testing, the majority of which focus on testing C++ systems. A further 22 entries, or 8%, are concerned specifically with Smalltalk. There is not one single entry related to both testing **and** Smalltalk.

The findings from the OOPSLA 95 "Testing Smalltalk Applications" workshop [Yates 95] give further evidence for the need for increased research related to testing Smalltalk systems. Some of the points raised by the workshop include:

- there are major differences in testing Smalltalk applications due to the role that the Smalltalk development environment plays in the testing process;
- a *disturbing* scarcity of literature and tools to support Smalltalk testing;
- large vendors, e.g. IBM and ParcPlace, have developed in-house test coverage tools but currently have no plans to turn these tools into products;
- recommended the use of test coverage analyzers.

Richard Bache and Martin Neil [Fenton 95] recommend the selection of a set of *well-established* metrics, as opposed to informally defining their own, when managers introduce metrics into an organization. Bache and Neil's set of six well-established metrics include test coverage metrics.

This paper will discuss what test coverage is, how it is used and describes different techniques for building a test coverage tool in Smalltalk. We reject techniques

*Level 1, 434 St. Kilda Rd, Melbourne VIC 3004, Australia; +61–3–9281–3765

†AI Dept, School of Computer Science & Engineering, P.O. Box 1, Kensington, Sydney NSW 2033, Australia; +61–2–9385–4034

based on bytecode interpreters due to their complexity, lack of documentation, and the non-standard nature of bytecode values in different versions of Smalltalk (see Section 3.1.1). We will endorse a text insertion technique that was initially inspired by a tool built by Murphy [Murphy 95b]. The simplicity of the endorsed approach is both a strength and a weakness. While an effective test coverage tool can be constructed quickly with this approach (see Section 3.1.2) we now believe that this approach has certain limitations (see Section 4). Extensions to this tool would require more elaborate parser technology.

2 Test Coverage

Test coverage is concerned with determining what proportion of a defined piece of computer code has actually been executed during a testing cycle. The aim of this analysis is to give an indication of which code was not executed by a test scheme and was therefore not tested. Test coverage does not give any indication of the quality of a testing scheme. Achieving 100% test coverage does not indicate that the application is error free. Instead it gives an indication of how much of an application was actually tested. The quality of the test suite is not been verified by test coverage.

The result of test coverage analysis is usually two fold:

- a metric indicating the proportion of code executed;
- data indicating which code was not executed.

The metrics produced by test coverage can be used as a quality indicator of the completeness of testing of a given piece of code. If test coverage produces a metric of 100% then some degree of confidence can be attributed to the test suite. Conversely, if the test coverage metric is only 35%, i.e. only 35% of the application code being tested was executed by the test suite, then this can be used as an indication that testing may not have been as thorough as expected. In this case the additional data produced by test coverage analysis would provide indications of which code was not tested.

2.1 Using a Test Coverage Analysis Tool

With the use of a test coverage analysis tool testing becomes an iterative procedure. The test suite is executed and a coverage metric is produced. If the metric is not high enough, i.e. not close enough to 100%, then the data detailing *uncovered* code is used to identify areas where addition tests are needed or the re-writing of existing tests is required. This cycle is then repeated until the coverage metric reaches the required target value. While the aim of testing would be a coverage metric of 100%, in reality this may not be achievable. The creation of test suites is no different from the creation of the application being tested, or any other form of computer programming, in that skill and effort is required to produce quality, error free code. The effort, time, resources and skill required to reach a coverage metric of 100% for a given application may not be feasible or achievable:

- theoretically, it is possible to analyze the logical flow network of a program and automatically generate a set of inputs that exercise all branches of a program [Zlatereva 92]. However, in the case where portions of the network are not known with certainty, then only very small programs can be so analysed [Menzies 96b]. Formally and empirically, it is known that the patterns of calls between methods in object-oriented languages that use pointers are always not known be certainty [Murphy 95a]. In order to demonstrate this, consider a message sent to a member of a container of `Shape` objects, where `Shape` can be one of `Triangle, Rectangle, Sphere`. Only in certain special cases[1] can a compile-time analysis uniquely resolve the receiver of this message. Also, in languages whose parameters are not strongly-typed, there is always some measure of indeterminacy in the logical flow network. Smalltalk suffers from both problems. While approximate method call patterns can be deduced from Smalltalk source code [Menzies 96a], these approximate method call patterns are not detailed enough to be used for auto-test generation;
- pragmatically, it may not be possible to (e.g.) simulate database failures or communication protocol errors;
- the construction of a complete test suite from the specification may be inhibited by two factors. Firstly, such an analysis is a non-trivial task

[1] e.g. when that message send is to a method that is defined only in one of `Triangle` or `Rectangle` or `Sphere`

which may require the assistance of scare project resources (e.g. user expert time). Secondly, in the case where the specification is incomplete (e.g. the typical OO evolutionary development), then a test suite deduced from the specification will always be incomplete.

2.2 Test Coverage and Smalltalk

There are several different test coverage metrics, each measuring different coverage attributes [Fenton 91, pages 183–185]. Some coverage metrics depend on the constructs defined by a particular programming language. Not all coverage metrics are applicable to all programming languages. Useful coverage metrics include the following:

Statement An indication of the proportion of code statements executed.

Branch Indicates the proportion of decision outcomes executed. For example, were both the true and false branches of an **if** statement executed.

Loop Indicates whether loops were tested with a range of invariants. For example, was a **for** loop tested with upper and lower bound invariants.

Boolean Expression Indicates the proportion of the conditional tests executed in complex decisions. For example, has the following expression been tested with values so that both sides of the **&&** operator have been exercised:

```
if ( X > 6 && Y <= 10 ) \{ ...
```

These definitions fit closely to languages such as C, Pascal and C++. Some modification is required to match them with the Smalltalk-80 language elements:

Statement Same as above *Statement* metric.

Conditional Selection A subset of the *Branch* metric. *Conditional Selection* deals with the execution of `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse` and `ifFalse:ifTrue:` messages.

Conditional Repetition Also a subset of the *Branch* metric. This metric deals with the `whileTrue`, `whileTrue:`, `whileFalse` and `whileFalse:` messages.

Fixed-Length Repetition This category is equivalent to the *Loop* metric. It deals with messages such as `timesRepeat:`, `do:`, `to:do` and `to:by:do`.

Logical Operations Equivalent to *Boolean Expression*. Concerned with the `&`, `|`, `not`, `eqv:`, `xor:`, `and:` and `or:` messages.

A major problem in defining the above coverage metrics for Smalltalk is that a user can add user-defined control structures to the system, since they are simply messages sent to objects. This means it would not be possible to fully define and implement some of these coverage metrics.

Murphy [Murphy 95b] defines four coverage metrics for Smalltalk; *Statement*, *Branch*, *Loop* and *Path*. The definitions are confusing and somewhat misleading, for the following reasons

1. Statement coverage is equated with the execution of a method, but a method can contain 0 to n statements.

2. No definition of a Smalltalk loop statement is given, apart from using the `do:` message as an example of one.

3. Murphy's definition of the Path metric attempts to measure whether "every logical path through a method has been tried" but is defined in terms of the number of possible paths available through *two* different methods.

Because of these ambiguities and difficulties it was decided to define two test coverage metrics for Smalltalk

Method Measures the proportion of executed instance methods in a Smalltalk Class. This is what Murphy intended, and demonstrated, for his *Statement* metric.

Block Measures the proportion of blocks executed in a given instance method. As a side effect this will also incorporate *Conditional Selection*, *Conditional Repetition* and *Fixed-Length Repetition* since all the messages in these metrics require a block.

3 TCAT: Test Coverage Analysis Tool for Visual Smalltalk

3.1 Design

This paper was inspired by certain perceived limitations in Murphy's Smalltalk test coverage tool [Murphy 95b]. Murphy uses the technique of *method wrapping* whereby the original method is *moved* to a new unused selector and a new method, with the original selector, is created. This new method performs the code required for coverage analysis and then sends a message, using the selector of the moved method, to invoke the original method. As an example of this technique the method

```
SomeClass>>aMethod: anObject
  "Send some messages to <anObject>"

  .....  some messages here .....
```

would be "replaced" by

```
SomeClass>>aMethod: anObject
  "Perform method coverage logging
  for SomeClass>>aMethod:"

  self
    log: #aMethod:
    forClass: SomeClass.
  "Call the original method"
  ^self real_aMethod: anObject
```

and a "new" method added to `SomeClass`

```
SomeClass>>real_aMethod: anObject
  "Send some messages to <anObject>"

  .....  some messages here .....
```

There are some limitations to Murphy's implementation (the first is noted by Murphy):

- binary messages, which are mostly used for arithmetic, cannot be wrapped due to the limitations placed on the selector name by the Smalltalk language [Goldberg 89]. A binary message selector is composed of one or two non-alphanumeric characters selected from the set {+/*-~<>=@%|&?!,}, with the added restriction that the second character cannot be a minus sign {-};
- after the methods in a class have been wrapped both the new methods added to perform the wrapping and the original, now renamed, methods will be visible in a class browser. This makes it difficult and confusing to move among the original class methods, which have been renamed, in the browser;
- to make a change to a method that has been wrapped would involve finding and altering the *renamed* method.

These limitations, and the desire to implement additional coverage metrics, were the impetus for finding a different approach to test coverage analysis in Smalltalk.

3.1.1 Bytecode Alteration

Initial investigations focused on the alteration of the bytecode data produced by the Smalltalk compiler. It would be theoretically possible to alter, referred to as *instrumenting*, this compiled code to perform additional processing to generate the data required for coverage analysis. This approach potentially offers the following benefits to method wrapping:

- the limitation of wrapping binary messages does not apply;
- the additional code added is not visible in method browsers;
- no additional methods are added to the class;
- potential improvements in the processing time required to instrument and remove instrumentation from methods as no code recompilation is required which is a resource intensive process.

The limitations imposed by this approach are:

- primitive methods cannot be instrumented as no code can be inserted prior to the primitive invocation. This may be a limited problem as primitive methods only appear in the Smalltalk base classes, which would not normally be tested in an application;
- vendor and compiler version dependence. Different vendors may represent instructions with different bytecode values and combinations of bytecodes. The test coverage tool would need to be re-implemented for each version of Smalltalk. Also a vendor may change the compiler and bytecode values in subsequent releases of their Smalltalk implementation, again requiring

re-implementation. This does not make for a portable tool.

To understand this approach some knowledge of how the Smalltalk language is implemented is required. While a detailed description of the implementation of Smalltalk can be found in [Goldberg 89] some of the salient points are included here for clarity. Smalltalk compiles source methods to bytecodes, eight bit numbers, which are interpreted by the stack-oriented Smalltalk virtual machine at execution time. The interpreter understands 256 bytecode instructions, 0 to 255, that can be categorized as pushes, stores, sends, returns and jumps. As more than 256 bytecodes are required to translate Smalltalk source, some bytecodes take extensions. An extension is simply an additional one, two or more bytecodes that further specify the instruction and it's parameters. In this manner a single virtual machine instruction can range from 1 to 4 bytecodes. In addition to the bytecodes the compiler also produces a set of objects referred to as the *literal frame*. The literal frame contains any objects that could not be referred to directly by bytecodes. The types of objects stored in the literal frame include

- global, class and pool shared variables;
- literal constants such as numbers, characters, strings, symbols and arrays;
- most message selectors.

Since the Smalltalk-80 language defined in [Goldberg 89] is not an industry standard different vendors choose to implement the language in slightly different ways. One notable area of difference between Smalltalk implementations is the compilation and execution of code. Some Smalltalk vendors include the source for the compiler in the base class library. The availability of the compiler source would assist in the process of mapping method source to bytecode values. Digitalk do not include the source code for the Visual Smalltalk 3.0.1 (VST) compiler. This means that determining the bytecode values for a particular segment of Smalltalk code is done by trial and error. After a great deal of experimentation the relationship between the bytecode values and some simple message sends was determined. The author was able to alter a methods bytecodes in such a way that an additional message send was inserted at the start of methods to store the executing method's name and class name in a global variable, for later analysis and reporting.

In VST compiled methods are stored in the class method dictionary as instances of `CompiledMethod`. One of the instance variables of `CompiledMethod` is `byteCodeArray` which, as the name implies, contains the bytecode values for the method as an array. The `CompiledMethod` class is a subclass of Array and, in addition to normal instance variables, stores the methods literal frame as indexed instance variables. The VST implementation generates the literal frame as two stack based lists growing from opposite ends and meeting in the center. Literal constants and shared variables are stored in parsed source order growing downwards from the *top* of the array. Message selectors are stored in parsed order, growing upwards from the end of the indexed instance variable array. An example showing the bytecodes and literal frame for some Smalltalk source can be seen in Figure 1. Bytecodes that refer to literal frame values do so by reference to their relative position in the literal frame stack. For example, in Figure 1 bytecode 226 refers to first message selector in the literal frame which happens to be the last element of the frame. Bytecode 227 refers to the second message selector which is the second last frame literal, and so on. For the constant/variable list 163 refers to the first constant element and 100 to the first shared variable element.

The data required for method level test coverage can be reduced to method name and class name. This requires the addition of two elements to the literal frame. Because of the two stack mechanism used by VST and the fixed length nature of the `CompiledMethod` object the steps required are as follows:

1. Determine the bytecode sequence for the message send we are going to insert into the method i.e.

```
self
    log: #methodName
    forClass: self class
```

2. Create a new instance of `CompiledMethod` of size 2 greater than the original method. Copy literal frame values from original method and add two new values (a symbol for `#method` (changed to the method selectors actual value) and an `Association` for the value of `self class`) at the start or end of the constant/shared literal frame list.

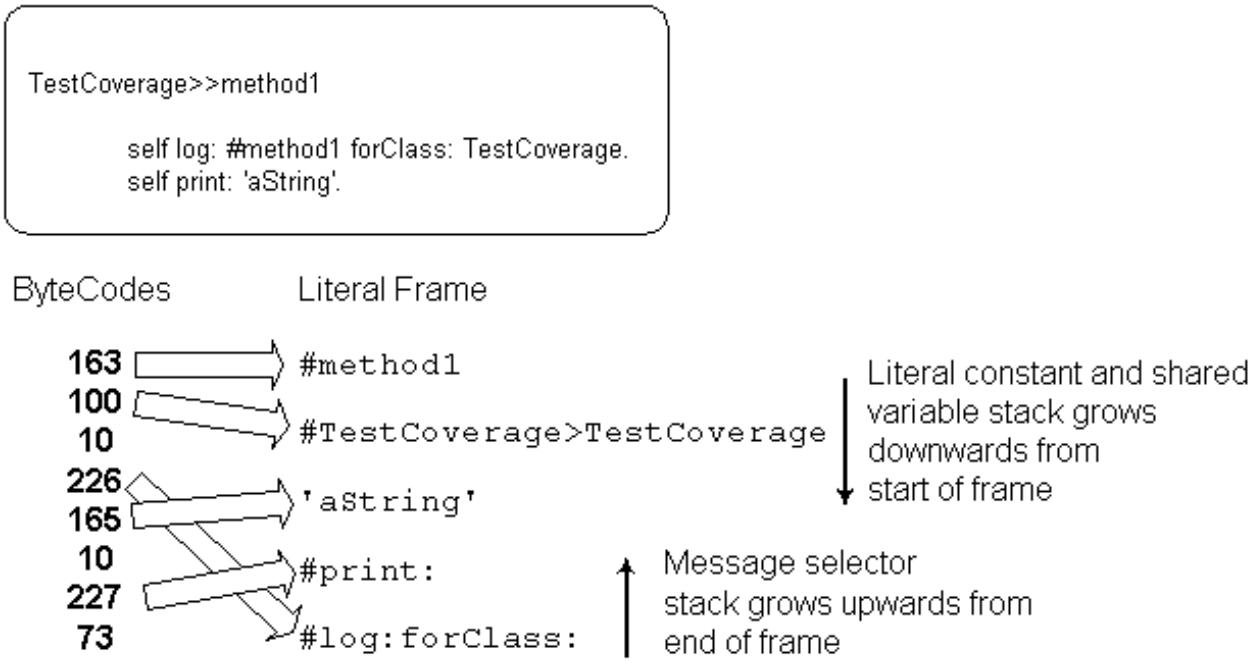

Figure 1: Relationship between Smalltalk source and Visual Smalltalk 3.0.1. compiled objects.

3. Insert the bytecode sequence from the first step above into the `CompiledMethod` instance variable `byteCodeArray` at the appropriate place.

4. Change all bytecode values that reference literal frame values to reference new positions in the literal frame since adding the two new values.

5. Copy the original `CompiledMethod` instance to a global variable and replace it with the new instance.

Step 4 turned out to be the most difficult. Without a definitive list of what every bytecode value means in the context of its surrounding bytecodes, determining which bytecodes need to be altered is difficult. After several attempts, we realized that even if we successfully worked out how to handle Step 4 for VST 3.0.1, we may have to change our system in order to handle the other bytecode systems used in Smalltalk implementations from other vendors or even newer versions of Digitalk Smalltalk[2]. Such changes are complicated by the lack of documentation on version-dependent bytecodes. Further, we believed that an improved version of Murphy's code insertion system would be simpler to implement and port to various Smalltalk systems. Hence, we abandoned bytecode insertion and moved on to a textual code insertion technique.

[2]Digitalk and ParcPlace recently merged to form ParcPlace-Digitalk

3.1.2 Code Insertion

The second approach proved more successful and is used in the implementation of TCAT. The basis of this approach is to parse the method source and insert additional code. The steps involved are as follows:

1. Determine the starting position in the method source text stream for insertion. This involves recognizing the methods message pattern, temporary variables and comments.

2. Create a string representing the new test coverage message send for this method.

3. Insert the new message send text into a copy of the method source text at the start point.

4. Use the VST compiler interface to compile the new method source, in the context of the methods class, producing a new `CompiledMethod` object.

5. Copy the original `CompiledMethod` object to a global variable so that it can be restored after testing has been completed.

6. Replace the original `CompiledMethod` with the new one in the methods class method dictionary.

7. Adjust the source code instance variable in the new `CompiledMethod` object to reference the original `CompiledMethod` object so that the original methods source will be displayed in Class and Method Browsers.

This approach provides Method level coverage as defined in Section 2.2. To implement the additional Block level coverage involves some more complicated parsing and the insertion of additional message sends following the start of each block. For Block coverage an additional parameter, being the block number in the method, is also recorded at execution time. This allows the identification of which block in a method was executed and the number of times. With this technique the method

```
SomeClass>>aMethod: anObject
  "Send some messages to <anObject>"

  self firstMessage: anObject.
  anObject isNil
    ifTrue: [ self output: anObject ]
```

becomes

```
SomeClass>>aMethod: anObject
  "Send some messages to <anObject>"

  TCMonitor logEvent: #aMethod:
            in: SomeClass.
  self firstMessage: anObject.
  anObject isNil
    ifTrue: [TCMonitor logBlockEvent: 1
             for: #aMethod
             in: SomeClass.
             self output: anObject ]
```

3.2 Implementation

TCAT, implemented in Visual Smalltalk 3.0.1. for Windows, consists of five classes:

TCMonitior This class handles the creation of instrumented methods, performs the monitoring, data gathering and metric reporting tasks for TCAT. It consists solely of class methods, no instance methods. Subclassed from `Object`.

TCInstrumentedMethod This class performs the tasks of creating an instrumented method for `TCMonitor`. The class is subclassed from `Object`.

TCViewer This class forms the main visual interface to TCAT. It allows for the selection of classes to be instrumented, the display of the coverage metrics, launching the TCAT Browser and restoring the original uninstrumented methods. It is subclassed from `ViewManager`.

TCBrowser This class is used for browsing the instrumented classes and is subclassed from the `ClassHierarchyBrowser`. In addition to showing method source it also shows the number of times methods and blocks in methods have been executed. `TCBrowser` also performs "on-the-fly" instrumentation of new and changed methods.

TCStack This is a "support" class subclassed from `OrderedCollection`. This class implements a simple stack data structure. Instances of this class are used by `TCInstrumentedMethod` in the parsing of source code.

To prevent problems caused by instrumenting a class more than once and the need for the inserted message send to store data in a global variable, the monitoring and data gathering function of TCAT is achieved with class methods not instance methods. This means that only one monitoring function is active in the image and can be accessed from any method in any class.

3.2.1 Using TCAT

Using TCAT is done by firstly starting the TCAT Viewer, `TCViewer`, with an optional collection of classes to be instrumented. The viewer opens with a hierarchical list of all classes in the top left-hand "source" pane and a list of classes to be, or being, instrumented in the top right-hand "instrumenting" pane. `TCViewer` can be started in three ways, for example

```
TCViewer open.
   or
TCViewer openOn:
    #( TestCoverageTests
        TestCoverageTests2 ).
   or
TCViewer openOn: #( 'TestCoverageT*' ).
```

The first example opens `TCViewer` with no entries in the instrumenting list, shown

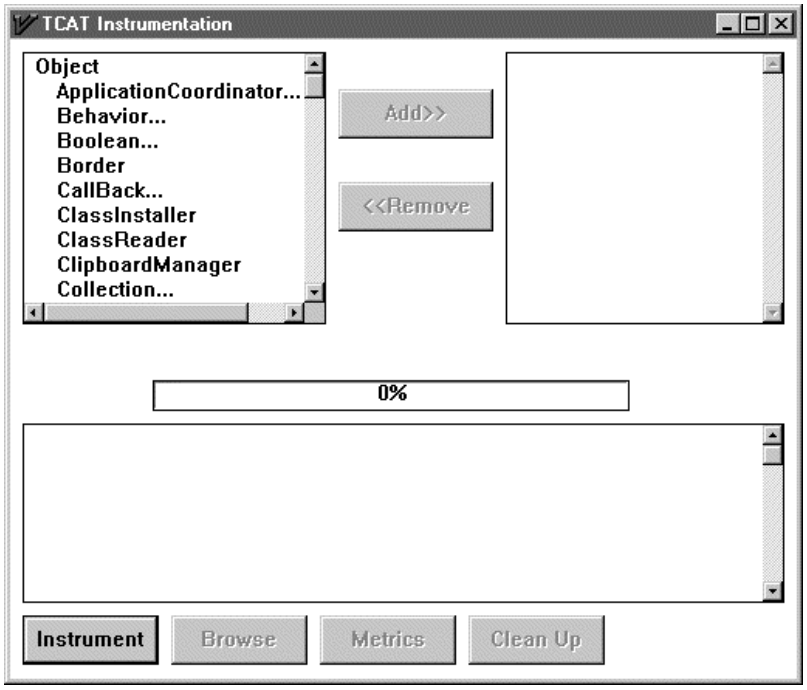

Figure 2: TCAT Viewer interface.

in Figure 2. The second opens `TCViewer` with the classes `TestCoverageTests` and `TestCoverageTests2` in the instrumenting list, as in Figure 3. The third example opens `TCViewer` with all classes whose name starts with the characters `TestCoverageT`. The third form is very useful if a common naming convention has been used for the classes in the application being tested. Which ever method of starting `TCViewer` is used classes can be added to the instrumenting list from the source list by selecting with the pointer and choosing the **Add**>> button. Similarly, classes can be removed from the instrumenting list by selecting them and choosing the <<**Remove** button.

When the classes to be monitored have been chosen selecting the **Instrument** button will start the process of method instrumentation. As each class is instrumented it's name appears below the source list and the sliding scale indicator shows what percentage of the chosen classes have been completed. If a method cannot be instrumented it's name appears in the scrolling window at the bottom of the window. This could happen if the method has no source code available or it is a primitive method. An example of this is shown in Figure 3. At this stage the chosen classes are instrumented and ready for testing. After testing has been completed selecting the **Metrics** button will display Method coverage and Block coverage metrics for the classes and an overall Method coverage and Block coverage metric for all the classes. This can be seen in Figure 4. The metric information is shown by class, in alphabetical class name order, and total for all classes. There are four figures shown in the metric data, each prefixed by a indicator character:

B This value shows the Block coverage metric. It consists of a proportion, the number of Blocks entered and the total number of Blocks in the class. A summary for all classes is also given.

C This value only appears in the "Total Coverage" line. It's value is the number of classes that were instrumented for test coverage.

M This value shows the Method coverage metric. It consists of a proportion, the number of Methods entered and the total number of Methods in the class excluding any Primitive Methods. A summary for all classes is also given.

P This value indicates the number of Primitive Methods in the class and is excluded if the number is zero. A summary for all classes is also given.

Additional coverage data is provided when the TCAT Browser is used. To us this Browser select the **Browse**

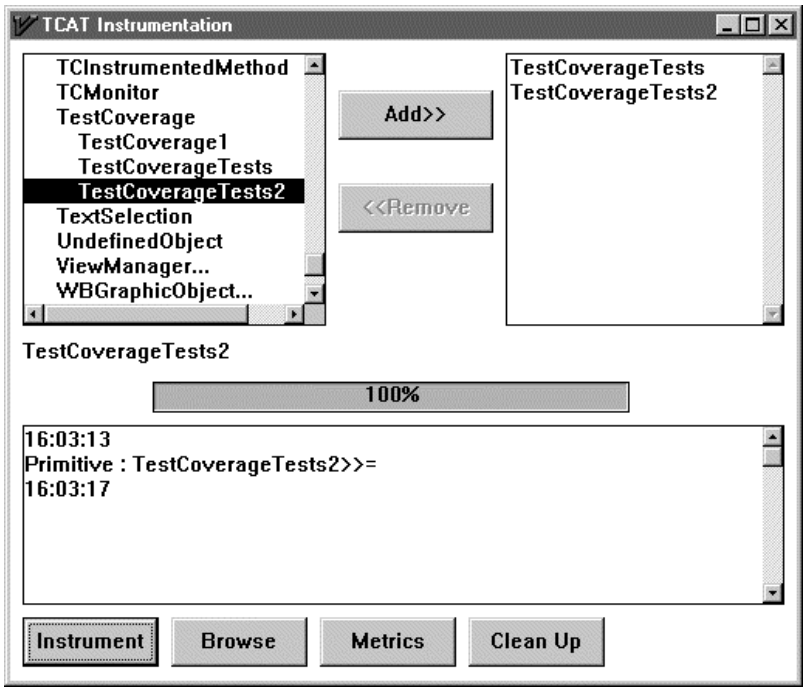

Figure 3: TCAT Viewer showing instrumented classes.

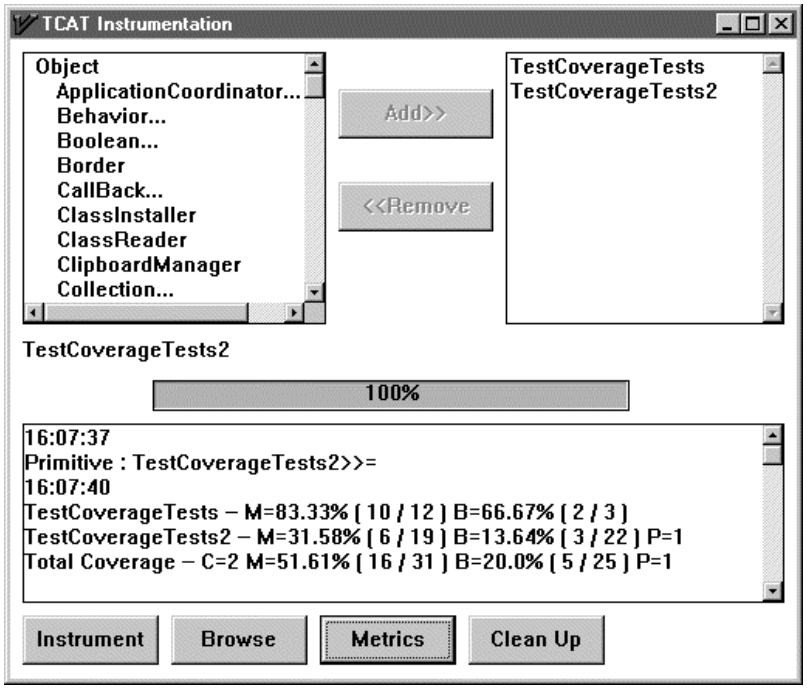

Figure 4: TCATViewer with metric data.

button. This will open a Browser similar to the familiar Class Hierarchy browser. The top left pane lists the classes that are being monitored for test coverage. The top center pane allows the selection of "Covered" or "Not Covered" methods via a pair of radio buttons. Below the buttons is a list of blocks in the currently selected method. The top right pane lists the methods in the selected class. The bottom pane shows the source for a selected method. The differences between the TCAT Browser and the standard Class Hierarchy Browser (CHB) are

- a number denoting the number of times a method has been executed appears after the method name in the method list pane. When "Not Covered" methods are viewed this number is removed since it is zero;
- the block list pane shows a pair of numbers, the first represents the block number and the second is the number of times that block has been executed. For the "Not Covered" list the execution number is always zero. For ease of reference blocks are numbered sequentially from 1 based on the order that they appear in the method;
- selecting a block in the block list causes the source code for that block to be highlighted in the source pane;
- various pane pop-up menu options have been removed;
- methods can be edited, deleted and new methods added as with a standard CHB. The TCAT Browser will instrument any methods which are added or changed.

To end the test coverage process the user selects the **Clean Up** button on the TCAT Viewer or closes the viewer window. These two actions restore the original uninstrumented methods.

4 Limitations of TCAT

None of the three limitations detailed earlier in Murphy's method wrapping technique (Section 3.1) apply to the code insertion technique used in TCAT. While the primitive method limitation (Section 3.1.1) for bytecode alteration is present in TCAT, the portability issue of the former technique is not.

A subtle drawback with the textual code insertion method used in TCAT is that while it can log the *entry* into a method, it cannot report the *exit* from a method. Consider the following code with the logging code inserted.

```
File>>emptyFile: aString
   | aStream |
   self log: #emptyFile:
        pos: 0 class: File class.
   aStream := self pathName: aString.
   ^( size := aStream size.
     size = 0
       ifTrue: [ self log: #emptyFile:
                    pos: 1
                    class: File class.
                 self remove: aString ].
     aStream close; release.
     size = 0 )
```

While we can tell when the method or its one block is entered, we do not know what happens after entry. For example, suppose the log reads:

```
File>>emptyFile: 0
File>>pathName: 0
Stream>>size 0
File>>emptyFile: 1
File>>remove: 0
Stream>>close 0
...
```

Using this log, we cannot tell if `File>>emptyFile:` or `File>>remove:` called `Stream>>close`. Hence, while we *can* say which methods were exercised, we *cannot* say in which order they were executed. Hence, TCAT cannot generate a *call-graph* representing the patterns of method calls. This is unfortunate since, if it could, then TCAT could have (e.g.) checked if the static call graphs generated for Smalltalk systems by Haynes and Menzies [Menzies 96a] represented true runtime behavior.

To obtain the data for call-graph analysis requires the knowledge not only of method/block entry, but also of when a method/block was exited. Suppose the above log read:

```
ENTER File>>emptyFile: 0
ENTER File>>emptyFile: 1
LEAVE File>>emptyFile: 1
ENTER File>>remove: 0
LEAVE File>>remove: 0
ENTER Stream>>close 0
...
```

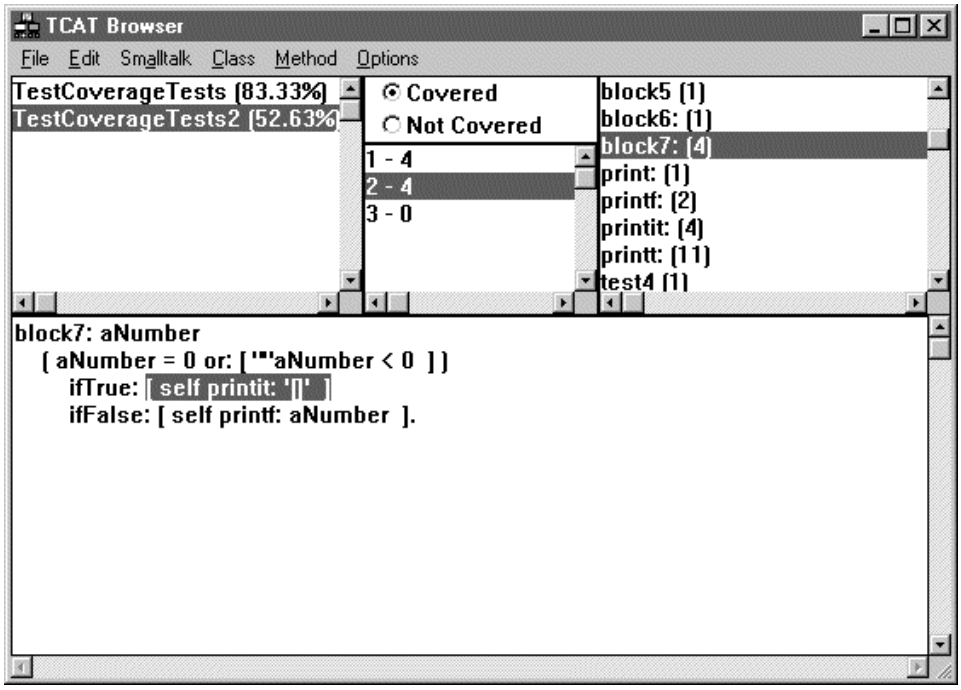

Figure 5: TCAT Browser showing Block metrics.

then there would no confusion about the order of method execution. Sadly, to log the LEAVE events, additional log code would have to be inserted at the end of blocks and methods. It is unclear how to do this without changing the return value of some methods. For example, in the implementation of `File>>emptyFile:`, if we place the LEAVE monitor just before the bracket on the last line, then the method will not return a Boolean. If we place it after the last bracket, then we would get a compiler error. A general LEAVE logging mechanism would have to find the last returned value, cache it, add a LEAVE log memo, then return the cached value. We attempted to sketch out how this might be done but found that there were too many special cases to consider.

5 Conclusion

We have discussed source code coverage tools in Smalltalk. We have cautioned that on theoretical and pragmatic grounds, 100% coverage should not be aimed for. However, our experience suggests that a majority coverage is a reasonable goal.

Three techniques for source code coverage tools in Smalltalk have been described:

- the method wrapping technique, as used by Murphy, which we sought to improve;
- the bytecode insertion system, which we abandoned since it was dependent on version-specific bytecode details;
- TCAT: our code insertion system, which can generate source code covering information but which cannot be used for discovering patterns of method calls since it cannot implement LEAVE logging.

In order to extend TCAT, we need LEAVE logging. A general LEAVE logging system would require access to the parse tree of the methods involved. We choose not to explore this options since many Smalltalk systems (e.g. VST) do not supply source code for their compilers. Hence, we conclude that:

- code insertion techniques, as used in TCAT, can be used to build useful code coverage tools. Note that monitoring and instrumenting functions of TCAT are a mere 600 lines long, including comments;
- LEAVE logging requires more powerful, but more complicated, parser-based systems.

The Smalltalk classes for TCAT can be obtained from `http://www.cse.unsw.edu.au/~timm/pub/lang/smalltalk/tcat`.

6 Acknowledgements

Philip Haynes, from Object Oriented Pty. Ltd., was our patient guide to the internals of the Smalltalk virtual machine.

References

[Fenton 95] Fenton, Norman, Whitty, Robin, & Iizuka, Yoshinori (eds). 1995. *Software Quality Assurance and Measurement: A Worldwide Perspective.* International Thomson Computer Press.

[Fenton 91] Fenton, Norman E. 1991. *Software Metrics: A rigorous approach*. Chapman & Hall.

[Goldberg 89] Goldberg, Adele, & Robson, David. 1989. *Smalltalk-80: The Language*. 2 edn. Addison Wesley.

[Menzies 96a] Menzies, Tim, & Haynes, Philip. 1996 (Jan.). *Empirical Observations of Class-level Encapsulation and Inheritance.* Tech. rept. Department of Software Development, Monash University, Caulfield, Melbourne VIC 3185, Australia.

[Menzies 96b] Menzies, T.J. 1996. On the Practicality of Abductive Validation. *In: ECAI '96.*

[Murphy 95a] Murphy, G.C., Notkin, D., & Lan, E.S.C. 1995. *An Empirical Study of Static Call Graph Extractors*. Tech. rept. TR95-8-01. Department of Computer Science & Engineering, University of Washington.

[Murphy 95b] Murphy, Mark L. 1995. Coverage analysis in Smalltalk. *The Smalltalk Report*, Oct., 4–8.

[Whitty 96a] Whitty, Robin. 1996a. Object-oriented metrics: A status report. *Object Expert*, Jan., 35–40.

[Whitty 96b] Whitty, Robin. 1996b (June). *Object-Oriented Metrics: an Anotated Bibliography.* Available from South Bank University web site at `http://www.sbu.ac.uk/~csse/publications/OOMetrics.html`.

[Yates 95] Yates, Barbara. 1995. Testing Smalltalk Applications: workshop report. *In: Addendum to the OOPSLA Proceedings.*

[Zlatereva 92] Zlatereva, N. 1992. Truth Maintenance Systems and Their Application for Verifying Expert System Knowledge Bases. *Artificial Intelligence Review*, **6**.