Branch Coverage for Arbitrary Languages Made Easy

Ira. D. Baxter, CTO
Semantic Designs, Inc.
12636 Research Blvd. #
Austin, Texas, 78579 USA
+1 512 250 1018
idbaxter@semdesigns.com

ABSTRACT
Branch coverage is an important measure of the thoroughness of testing. One can easily get tools that collect this information for mainstream languages (C, Ada) on mainstream platforms (Solaris, UNIX). Such tools are difficult to find for less widely used or interpretive languages (JavaScript) or languages used on nonstandard platforms (C in embedded systems).

This paper shows the straightforward result that an industrial strength source-to-source transformation system can install test probes in software systems easily. What is not obvious is that such transformation systems exist. The consequential good news is that branch coverage testing tools can be easily built for all kinds of software in all kinds of execution environments.

Keywords
Test coverage, branch coverage, transformations

1 HOW CAN WE EASILY INSTRUMENT CODE?

Testing software is hard. Knowing that software is well tested is even more difficult. One widely accepted measure is that of branch coverage, the ratio of basic code blocks that were exercised by some test, to the total number of code blocks in the system under test. In principle, this is easy to obtain: instrument each basic block in a copy of the source code with a probe, and accumulate probe information across a set of tests.

In practice, doing the instrumentation is hard, because reliably identifying the basic blocks is difficult due to variation of syntax across languages. Two principal means exist: modifying object code, and modifying source code.

Modifying source code is conceptually straightforward: identify each basic block in the program, and insert a line of code acting as a self-identifying probe in that basic block. Figure 1 shows an instrumented C program, with the probes italicized. Each basic block gets a unique identifying number, associated with it source file and line number. Hidden boilerplate initialization code resets all the visited flags; running the program sets them, and some hidden auxiliary support collects and integrates the results of the visited vector across multiple tests. Additionally, this acquires the extremely useful information about what part of a system has not been executed by any test; such code may not be reliable.

Identifying such basic blocks essentially requires that the program be parsed, analyzed semantically for basic blocks, and that the source be modified with corresponding probes. In essence, one needs a complete language front-end to do this. This is a daunting job for most practical languages. (Solutions involving PERL-like string-manipulation tools are not robust in the face of lexical rules, conditional compilation, etc.)

Modifying object code trades the problem of understanding language syntax for the problem of understanding and patching machine code (actually, link editor code) for a particular platform, and being able to correlate machine instructions with source lines. A nice benefit of this approach is that test coverage is implemented across essentially all the languages having compilers for that

```
int fibcache[1000];  // initially 0s

int fib(int i) // fast Fibonacci
{ int t;
  visited[1]=1;
  switch (i)
  { case 0: visited[2]=1;
    case 1: visited[3]=1;
            return 1;
    default:
      visited[4]=1;
      if (fibcache(i))
         { visited[5]=1;
           return fibcache(i);}
      else { visited[6]=1;
             t=fib(i-1);
             fibcache(i)=t+fib(i-2)
             return fibcache(i);
           };
   };
   visited[7]=1;
};
```

Figure 1

platform. Techniques for doing this have been implemented on many mainstream platforms because the cost can be amortized.

Unfortunately, this method is not available for nonstandard or interpretive languages (e.g. JavaScript), nor is it available for nonstandard execution environments (embedded C). Yet a huge amount of software is implemented this way. This leads to a situation in which one cannot obtain a desired measure of test quality for large classes of software.

In essence, the only real solution is the source probe installation scheme. One impractical cure is to persuade every language vendor to add probe installation to his language front end. A possible cure is for an organization to find a compiler toolkit, and use it to achieve probe coverage. This requires considerable skill and effort on the part of the testing organization to a technology that is not mainstream to their activity, and so this rarely occurs.

A more practical cure is to use a source-to-source transformation system to install such probes. Arguably, this also requires considerable skills, but significantly less skill and time is needed to achieve an effective result. Then a software engineering organization can take its test management into its own hands, by writing a small number of transforms. While this appears to be a relatively obvious result, we see few organizations actually doing it, so this paper exhibits the basic technique.

Given such tools, the transforms required to insert probes are straightforward to write and use.

The paper is organized roughly as follows:

1. Discussion of the nature of an industrial strength transformation system
2. Examples of transformations for test coverage for the C language
3. Discussion of additional infrastructure needed to implement test coverage

2 INDUSTRIAL-STRENGTH TRANSFORMATION SYSTEMS

By industrial strength transformation system, we mean ones that:

- Accept language definitions for real languages
- Accept source-to-source rewrite rules in those languages
- Can apply those rules to a source base reliably
- Are available on commodity platforms

We know of only a few such systems, but they do exist:

- REFINE (available commercially from Reasoning Systems: www.reasoning.com)
- XT (available as a research tool via www.program-transformation.org/xt)
- DMS (available commercially from Semantic Designs: www.semdesigns.com)

These tend to be commercial systems because of the effort it takes to implement them. There are a host of other transformation systems, many listed at www.program-transformation.org. (The author apologizes to any that might meet the criteria but are not listed here, and would appreciate knowing about them.)

Many compiler toolkits (e.g. YACC) offer LL(1) or LALR(1) parsers, which work by definition only for very limited classes of languages. Most compilers and tools tend to have parsers with ad-hoc modifications to step around the limitations. This means the compiler infrastructure is not good for a wide range of languages. What one needs is a full context-free parsing mechanism, which both XT and DMS have (both systems use Tomita style parsers [7,8])

What distinguishes industrial-strength transformation systems from compiler toolkits is: the configurability and robustness of their parsing technology, the integration of that parsing technology with the pattern languages used for source-to-source rewriting, and the ability to regenerate legal source programs in all details from ASTs. This capability is used for large scale reengineering (e.g., code porting), software quality analysis and enhancement (e.g., clone detection and removal [4]), reverse engineering [2], etc. For DMS, this integration serves to support a long-term goal of design management [1,3].

To use these transformation systems, the language syntax of interest has to be defined. Because these tools are highly configurable, this is far less of a task than building a compiler front end. Further, these systems are often available with predefined language modules for mainstream languages, such as C, C++, Ada, FORTRAN, etc.

3 SOURCE TO SOURCE TRANSFORMATIONS

Source to source program transformations ("rewrite rules") are used to modify programs directly in terms of the programming language syntax. (Other program transformations may be implemented by procedural code, or sets of transformations). These rewrites are usually stated in terms derived from an abstract or concrete grammar, and these terms in turn correspond to underlying abstract syntax trees.

A typical rewrite rule abstractly has the following form:

$$LHS \rightarrow RHS \ \textbf{if} \ condition$$

where both *LHS* ("left hand side") and *RHS* ("right hand side") represent source language patterns with variables to represent arbitrarily long well formed language sub strings. The **if** *condition* is an optional phrase referring to the variables in the *LHS* pattern. These rules are interpreted as, "when a program part matches the *LHS*, replace it by the *RHS*, **if** *condition* is true". The *condition* may be implemented as some additional matching constraints, or a call on some decision procedure.

Real transformations systems add more syntax to this simple scheme to allow specification of more details about the patterns. For DMS, an example rewrite on C code to convert an assignment statement into an auto-increment is shown in Figure 2. This rule is written in DMS's *Rule Specification Language*. (For these examples, we take advantage of the availability of a C language module for DMS. We also take slight liberties with the transforms to simplify their presentation).

```
default domain C.

rule auto_inc(v:lvalue):
    statement->statement =
  “\v = \v+1;” rewrites to “\v++;”
       if no_side_effects(v).
```

Figure 2: A DMS rewrite rule

This defines a rewrite `rule` with name `auto_inc`, having a syntax variable `v` of syntactic class `lvalue`. The rule is a map from a C `statement` to a C `statement` (maps from one syntax class to another, and maps from one language to another are also possible with DMS). The text inside the quote marks is legal C source, modulo the possibility of escaped rule language tokens (marked with `\`), such as syntax variables (e.g., `\v`), which stand for arbitrary legal C source. The left hand quoted string is the rule *LHS*, and the right hand quoted string is the *RHS*. The *LHS* represents any legal C `lvalue`, and the *RHS* represents any legal expression adding some lvalue and one. The occurrence of the same syntax variable multiple times in the *LHS* requires that the same exact sequence occur in both places; this is how the rule is constrained to match identical source and target. The occurrence of the syntax variable in the *RHS* requires that the changed program include what was matched for that variable on the *LHS*. Finally, the **if** *condition* in this rule is a language-dependent decision rule that makes sure that the program fragment matched by `\v` contains no side effects, which would make this transformation incorrect.

before: `(*Z)[a>>2]=(*Z)[a>>2]+1;`

after: `(*Z)[a>>2]++;`

Figure 3

Before rule use, a typical rewriting engine first parses the rule according to its rule language, and then parses the quoted pattern in the language to be transformed (here specified by the `default domain` phrase as the “C” language), to construct pattern trees. At transformation time, the rewrite engine matches the *LHS* pattern tree against portions of the program, and replaces matched trees by the corresponding *RHS* tree if the condition is satisfied. The Figure 2 rule has the effect shown in figure 3.

Typically a transformation system will have a large number of rules, and a large number of possible places in a program to apply them. It is beyond the scope of this paper to describe how the transformation system chooses which rules and where to apply them. The simple notion that all rules possessed are applied leaf-upwards to the entire parse tree for a file is adequate for this paper, and supported directly as one mode of operation of DMS.

4 REWRITES FOR TEST COVERAGE

Figure 4 shows a few of the rewrite rules required to put instrumentation in C programs. Such rules are easy to define for procedural languages, because such languages

```
external pattern new_place
      (x:statement_sequence).

rule mark_function_entry
   (result:type,
    name:identifier,
    decls:declaration_list,
    stmts:statement_sequence) =
 “\result \name { \decls \stmts };”
       rewrites to
 “\result \name
    { \decls
      { visited[\new_place\(\stmts\)]=1;
        \stmts }};”.

rule mark_if_then_else
   (condition:expression;
    tstmt:statement;estmt:statement) =
 “if (\condition)\tstmt else \estmt;”
      rewrites to
 “if (\condition)
    { visited[\new_place\(\tstmt\)]=1;
      \tstmt}
  else {visited[\new_place\(\estmt\)]=1;
        \estmt};”.

rule mark_switch_case
  (condition:expression,
   stmts:statement_sequence) =
 “case \e: \stmts”
    rewrites to
   “case \e:
     { visited[\new_place\(\stmts\)]=1;
       \stmts }”.
```

Figure 4

mark all points of control transfer with explicit syntax. One needs essentially one transform per control-transfer syntax fragment.

There are two slightly tricky parts of these transforms. The first is the introduction of a new place number for each transform application. The tree-producing function `new_place` invents a new number each time it is called, and associates it with the source file and line number of each tree passed to it. (The DMS parsing infrastructure stamps every tree with source information, making this straightforward).

To do this for an entire software system, the tool must invent new place names that are unique across all source files involved in test. DMS can read tens of thousands of files in a single session, apply these transformations, and then produce the complete set of modified files.

```
if (condition)
  {  x=y;
     return;
  }
following code
```

Figure 5: Control transfer from conditional block

The second issue is installing probes in the code following a conditional block that contains control transfers. When a control transfer is found in a conditional block (Figure 5), a probe must be inserted before the statement following the conditional (Figure 6). This takes a small set of rules to "push" knowledge of the control-transfer up to the containing conditional, and then install the probe. In the interest of brevity, details of these transforms are not provided here.

```
if (condition)
  {  x=y;
     return;
  }
visited[place]=true;
following code
```

Figure 6: Inserted probe after conditional block

Applying these transforms to an undecorated source program produced the result in Figure 1. It is easy to invent other interesting types of probes. Simply revising the transforms to increment `visited` slots changes the probes from test coverage to profiling probes.

5 ADDITIONAL INFRASTRUCTURE NEEDED

Along with the probes, some additional support is needed to make test coverage useful. This code is all straightforward to produce manually.

First, some additional code is required in the system under test. This code may need to be hand (or script) patched into the source in a few places:

- A single-line declaration of the `visited` array. The transformation tool can report the largest new *place number* as the array size after installing probes.
- A tiny initializer loop that resets the `visited` array when the systems under test starts up.
- A collector mechanism, that writes the `visited` array result out to some accumulating engine, when the probed program terminates. In an embedded system, this probably writes the `visited` information to some external development system.

Second, an accumulating engine must accumulate the results of multiple tests. For test coverage, this is simply ORing the last collected `visited` array element-wise into a `was_visited_by_some_test` array. One could also record the test number that visited a place to provide finer reporting detail. For profiling, the `visited` array is simply added element-wise to the accumulated result.

Third, a cross-reference between place names and source files will be required to interpret the coverage results. DMS can be straightforwardly configured into emitting the collected association after inserting probes in all the files.

Lastly, some simple tool is needed to display the `was_visited_by_some_test` array, and another to accumulate statistics on a per system, per file basis is probably useful. One could harness DMS's ability to pretty print files as HTML, to color code visited/not-visited places differently, providing a visual display.

All of the additional infrastructure should easily be within the scope and capacity of the testing group for an organization to construct and maintain.

We have built such infrastructure, and implemented a display tool in Java (Figure 7). The display tool provides display of any selected file, showing lines where probes have been installed and the coverage status of that probe. Useful additions include to the display tool include a boolean bit vector calculator to not only OR test vectors together, but to compute AND and AND NOT, enabling a tester to easily determine which sets of tests overlap, and which tests in one test set are redundant with respect to other test sets. Finally, one can produce summary information and per-file coverage information as a report for management and record-keeping.

6 EXPERIENCE

This technology has been used to construct production test coverage tools for:

- ANSI 89 C. A special front end is used to handle he preprocessor directives. We expect other dialects such as GNU C, Microsoft Visual C++ and ANSI 99 C to be simple extensions. C++ requires more rules but the basic issues are otherwise identical.
- ANSI COBOL 85
- Java 1.1 and Java 1.3
- PARLANSE (a parallel language used to implement DMS itself)

Each language requires its own set of language-specific probe-installation transforms, but they all share the Java coverage display tool.

Very large systems have been tested, including a Java application of over 3500 source files requiring 77000 probes (the image in Figure 5 was taken from this case). Probe overhead in very tight loops is around 50%; across an entire application, it tends to be around 15%.

7 OTHER POSSIBLE TEST APPLICATIONS

Industrial-strength program transformation systems may be used for a number of other possible testing applications.

One previous approach [6] has used Refine to enhance mutation testing by generating mutant programs from the program source.

Path coverage is a better indication of testing completeness than test coverage, but the number of syntactically legal paths in a program is generally enormous and therefore untestable in practice. However, many syntactically legal paths are semantically impossible due to language or application constraints. By using symbolic reasoning to compute path prefix conditions and eliminating paths whose prefix condition is false, [5] limited the number of actual feasible paths through a large Ada program to a few hundred thousand. One possible research avenue would use program transformation systems to read applications, determine feasible paths, and install probes to verify whether all feasible paths had actually been executed.

A different but very useful approach would be to transformationally "compile" specifications into test case generators. Building on that idea, if one coupled symbolic execution with test case generation and feasible path analysis, one might achieve a means of finding bugs in code by analysis.

The ability to insert probes directly in code could also be used to instrument code for run time status collection, and/or automated data serialization to support data transmission or long-term storage of state information.

8 SUMMARY

Test code coverage is an important measure of quality for software systems. Obtaining coverage information for non-mainstream languages, or for programs in nonstandard execution environments has traditionally been difficult, as it requires tricky object code patching technology, or complex parsing infrastructure.

Industrial strength transformation systems can make it straightforward to implement test coverage in these unique circumstances, by:

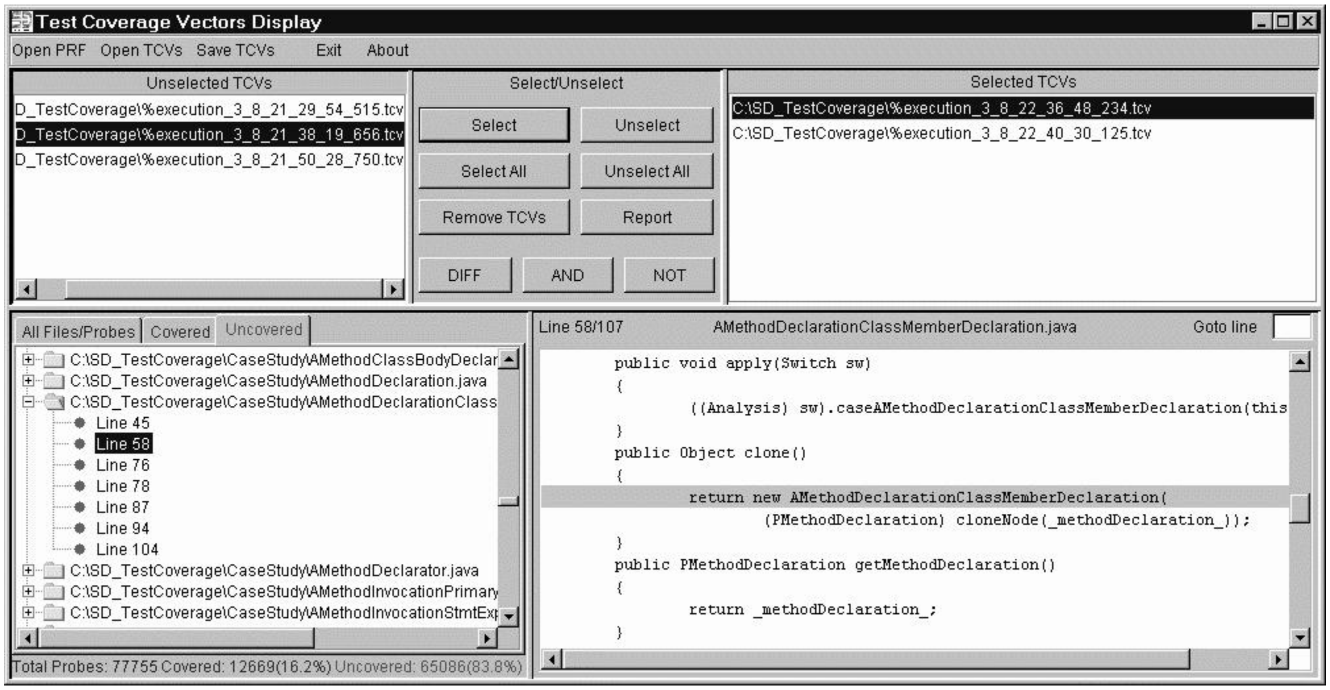
- Defining the language of interest to the tool (sometimes already available off-the-shelf)
- Writing a small set of source-to-source transformations.
- Implementing a small set of additional support procedures to aid statistics initialization, accumulation, display and analysis.

We have shown how to build practical test coverage tools for unique languages and environments. This enables the collection of good test quality statistics, and crucial indications of untested code, for large classes of applications for which such tools were previously unavailable.

More details can be found at *www.semdesigns.com*.

REFERENCES

1. Baxter, I. Design Maintenance Systems, *Comm. of the ACM* 35(4), 1992, ACM.
2. Baxter, I. and Mehlich, M. Reverse Engineering is Reverse Forward Engineering. 4th Working Conference on Reverse Engineering, 1997, IEEE.
3. Baxter, I. and Pidgeon, C. Software Change Through Design Maintenance. International Conference on Software Maintenance, 1997, IEEE.
4. Baxter, I, et al. Clone Detection Using Abstract Syntax Trees, International Conference on Software Maintenance, 1998, IEEE.
5. Goldberg, A, Wang, T. C., and Zimmerman, D. Applications of feasible path analysis to program testing, Proceedings of International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA August, 1994
6. Kotik, G and Markosian, L. Automating Software Analysis and Testing Using a Program Transformation System. SIGSOFT Software Engineering Notes 14(8):75-84, 1989, IEEE.
7. Tomita, M. Efficient Parsing for Natural Languages, 1988, Kluwer Academic Publishers.
8. van den Brand, M., et al. Current Parsing Techniques in Software Renovation Considered Harmful, Sixth International Workshop on Program Comprehension, 1998, IEEE.



```
D_TestCoverage\%execution_3_8_21_29_54_515.tcv
D_TestCoverage\%execution_3_8_21_38_19_656.tcv
D_TestCoverage\%execution_3_8_21_50_28_750.tcv
```

```
C:\SD_TestCoverage\%execution_3_8_22_36_48_234.tcv
C:\SD_TestCoverage\%execution_3_8_22_40_30_125.tcv
```

```
C:\SD_TestCoverage\CaseStudy\AMethodClassBodyDeclar
C:\SD_TestCoverage\CaseStudy\AMethodDeclaration.java
C:\SD_TestCoverage\CaseStudy\AMethodDeclarationClass
    Line 45
    Line 58
    Line 76
    Line 78
    Line 87
    Line 94
    Line 104
C:\SD_TestCoverage\CaseStudy\AMethodDeclarator.java
C:\SD_TestCoverage\CaseStudy\AMethodInvocationPrimary
C:\SD_TestCoverage\CaseStudy\AMethodInvocationStmtExp
```

Total Probes: 77755 Covered: 12669(16.2%) Uncovered: 65086(83.8%)

Line 58/107   AMethodDeclarationClassMemberDeclaration.java

```
        public void apply(Switch sw)
        {
                ((Analysis) sw).caseAMethodDeclarationClassMemberDeclaration(this
        }
        public Object clone()
        {
                return new AMethodDeclarationClassMemberDeclaration(
                        (PMethodDeclaration) cloneNode(_methodDeclaration_));
        }
        public PMethodDeclaration getMethodDeclaration()
        {
                return _methodDeclaration_;
        }
```

Figure 7: Generic Test Coverage Display Tool