Automatic Generation of Test-Cases for Software Testing

Gary McGraw and **Christoph Michael**
Reliable Software Technologies Corporation
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
`{gem,ccmich}@rstcorp.com` `http://www.rstcorp.com`

Introduction

In software testing, one is often interested in judging how well a series of test inputs tests a piece of code — the main idea being to uncover as many faults as possible with a potent set of tests. Unfortunately, it is almost impossible to say quantitatively how many "potential faults" are uncovered by a test set, not only because of the diversity of the faults themselves, but because the very concept of a "fault" is only vaguely defined (Friedman & Voas, 1995). This has lead to the development of *test adequacy criteria*, criteria that are believed to distinguish good test sets from bad ones. When a test adequacy criterion has been selected, the question that arises next is how one should go about creating a test set that is "good" with respect to that criterion. That question is the topic of this abstract.

We are specifically concerned with adequacy criteria that require certain features of a program's source code to be exercised. A simple example would be a criterion that says, "Each statement in the program should be executed at least once when the program is tested." Test methodologies that use such criteria are usually called *coverage tests*, because certain features of the source code are to be "covered" by the tests.[1]

The example given above describes *statement coverage*, which is a coverage criterion not in general use. The simplest coverage criterion that *is* used in practice is *branch coverage*. This criterion requires that every conditional branch in the program must be taken at least once. For example, to obtain branch coverage of the code fragment:

```
if (a >= b) { do one thing }
else        { do something else }
```

requires one program input that causes the value of the variable **a** to be greater than or equal to the value of **b**, and another that causes the value of **a** to be less than that of **b**. One effect of this requirement is to ensure that the "do one thing" and "do another thing" sections of the program are both executed.

There is a hierarchy of increasingly complex coverage criteria having to do with the conditional statements in a program. At the top of the hierarchy is *multiple condition coverage*, which requires the tester to ensure that every permutation of values for the boolean variables in a condition occurs at least once.

With any of these coverage criteria, the question arises of what to do when a test set fails to meet the chosen criterion. Often, the next step is to try to find a test set that *does* satisfy the criterion, but this can be quite difficult because the condition to be covered may be deeply nested in the code, and it is necessary, in essence, to execute the program backwards in order to discover which inputs will cause the criterion to be met. For instance, in the branch coverage example given above, it might be necessary to find a set of inputs that cause **a** to be less than **b**. It is easy to demonstrate that finding a set of tests that satisfy multiple condition coverage (as well as most other coverage criteria) is equivalent to solving the halting problem; there is no algorithm that can perform this task successfully in all cases. We suggest that a heuristic approach may hold promise.

In particular, we are investigating the application of case based reasoning (CBR) to the automatic generation of test cases by applying a set of pre-fabricated tweaks to user-defined test-cases (Kolodner, 1993; Turner, 1992). Preliminary work is focused on understanding what strategies human testers resort to when confronted with an initially-poor test set. Research into these methods will guide the creation of a set of tweaks that could in-principle impart a modicum of creativity to a test-set-generation program (Schank & Leake, 1989).

As part of test-case generation, our research effort must by necessity concentrate on tracing backwards through a program. In the example above, for instance, our program is required to automatically devise a way of making sure that **a** turns out to be less than **b** in at least one test case. We are also focusing on capturing the inherent "structure" of test cases. CBR should prove especially apropos to this aspect of the task.

[1]There is no reason to believe that the methodology described in this abstract will not work for other kinds of adequacy criteria as well, but certain flavors of coverage testing are of immediate interest because the Federal Aviation Administration requires that safety-critical aviation software be tested with inputs satisfying *multiple condition coverage*, a particular type of coverage test.

References

Friedman, M. and Voas, J. (1995). *Software Assessment: Reliability, Safety, Testability*. New York, NY: John Wiley & Sons.

Kolodner, J. (1994). *Case-Based Reasoning*. San Mateo, CA: Morgan Kaufman Publishers.

Schank, R. and Leake, D. (1989). Creativity and learning in a case-based explainer. *Artificial Intelligence*, 40(1-3),353-385.

Turner, S. (1992). MINSTREL: A Computer Model of Creativity and Storytelling. Doctoral dissertation. Los Angeles: UCLA.