A modular tool for automated coverage in software testing

Eugenia Díaz, Javier Tuya, Raquel Blanco

Department of Computer Science, University of Oviedo
Campus de Viesques s/n, Gijón, Asturias
33203 Spain
[eugenia, tuya, rblanco]@lsi.uniovi.es

Abstract. Software testing is an expensive and difficult process which need much time. For this reason, the existence of tools that allow to decrease this effort is very important. Our tool automatically generates test cases in order to obtain branch coverage in software testing from a source code. All process is automatic (source code instrumentation and test cases generation) and therefore the total time used in software testing is reduced. We describe the modules of the tool and present the result we have obtained compared the needed time to generate the test cases with manual instrumentation and the needed time with an automatic process.
Keywords. Automated software testing, software testing tools, automated instrumentation, branch coverage, tabu search.

1. Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in software development [1]. With techniques for automating the generation of software testing, we will be able to test the software more efficiently while reducing the time taken up by this task, thus reducing the cost and increasing the quality of the final product.

Although there are a lot of testing techniques [1], the important "craft" aspect that they entail has produced a natural interest in the automation of the process of creating software test cases. Among the different approaches used for the automation of this process, we may distinguish between random techniques [12] (test data are generated randomly to cover the input variables domains as much as possible), static techniques [4] (the program under test is not executed) and dynamic techniques [9] (which carry out a direct search of the test data through the execution of the program, which has to be previously instrumented).

There are several types of tools in order to facilitate the software testing process, and they have different functionalities. Among these functionalities we can find the following ones: to automate the path achieved in source code by a test case [2], to automate the execution of software tests [13] and to automate the generation of test cases by means of the instrumentation of the source code under test [3], [5], [10], [11], [14]. This instrumentation can be in automatic way or by hand.

Our tool allows to generate automatically branch coverage software test cases for programs written in C/C++. This automation affects to the source code instrumentation and to the test case generation. Besides, it allows to use several test cases generators, being independent of them.

2. Automatic tool description

The implemented tool has several modules:

- Parser: it generates the control flow graph of the source code under test.
- Instrumenter: it generates the instrumented source code.
- Test cases generator: it generates the test cases, using the instrumented source code and the control flow graph.

The scheme of our tool appear in Figure 1.

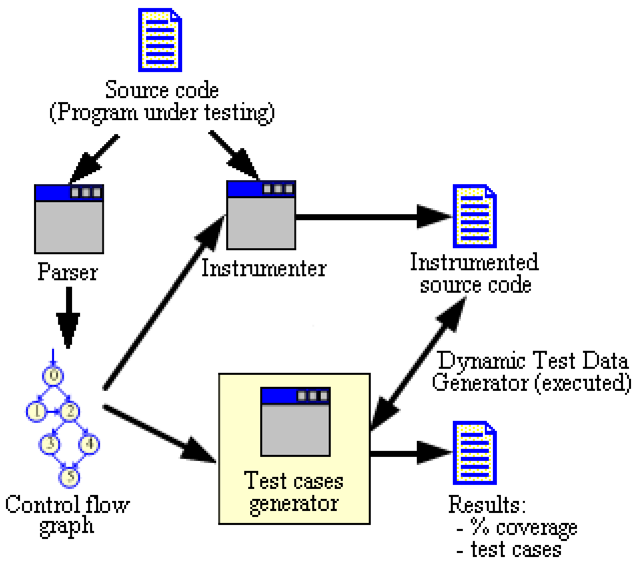


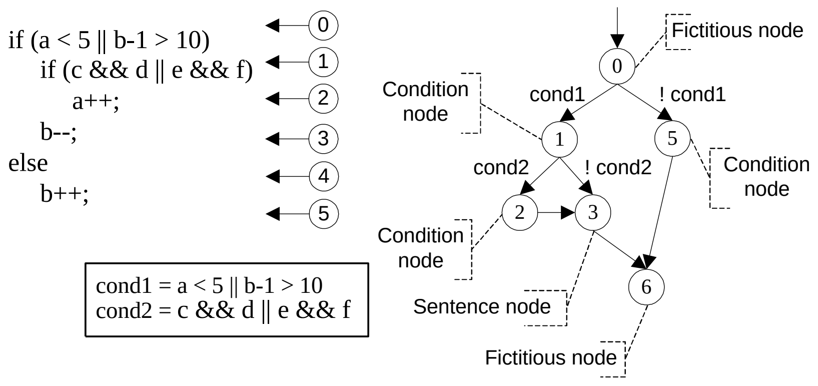
Figure 1. Automatic Tool Scheme

A parser has been developed that generates a file with the control flow graph from the source code of the program that is going to be tested. Each graph node stores important information that is used in the testing process. The instrumenter then reads the source file and instruments the program under test using the control flow graph. Finally, the test case generator is executed from the instrumented source code and its complexity graph: in each iteration it generates test cases for the program under test and executes it with them to store their behavior. The generator finishes when it obtains a desired branch coverage percentage or reaches the maximum number of attempts allowed.

2.1 Parser

The parser carries out the control flow graph, following the language grammar (C++) of the program under test.

Figure 2 shows an example of a complexity graph construction. A complexity graph has at least one node. This node is node 0 and marks the beginning of the control flow. When an if sentence is found in the program, a new node with the condition is created. This node represents the content of the if part. When the else part is detected, another node is created, whose condition corresponds to the negation of the condition. This node represents the content of the else part. Finally, a fictitious node is created to close the if-else block, when its end is detected. If the conditional statement has an if part and an else part, the node that closes the block joins the two branches corresponding to both parts. If the statement has not else part, the final node joins the branch of the if part to the initial node of this branch. Thus, it is possible to arrive at the node if the condition is true or false. This last node is a sentence node.



```
if (a < 5 || b-1 > 10)        ← 0
    if (c && d || e && f)     ← 1
        a++;                  ← 2
    b--;                      ← 3
else                          ← 4
    b++;                      ← 5
```

```
cond1 = a < 5 || b-1 > 10
cond2 = c && d || e && f
```

Figure 2. Example of graph generation.

The information that the test case generator needs to work correctly is stored during the generation of the control flow graph. This information consists of the best solution that a node reaches and its cost and the best solution that its children could reach and their respective cost. The conditions of the nodes, the best solutions and the costs are recovered during the testing process.

2.2 Instrumenter

The instrumenter produce the instrumented code that the test cases generator needs for its execution. This module reads the source code, and using the control flow graph, changes the conditions of the control flow statements and inserts additional instructions, which are known by the test case generator.

When the instrumented code of a condition is generated, the condition includes a transformation in which the relational operations turn into arithmetic expressions that are stored in a stack called the expressions stack. The new condition consists of arithmetic expressions, related to logic operators. Each expression, logic operator and bracket occupies a position in the expressions stack. Figure 3 shows the transformation of the relational expressions. The contents of the expressions stack are defined by means of a grammar according to which the code for each AND subexpression and OR subexpression are generated.

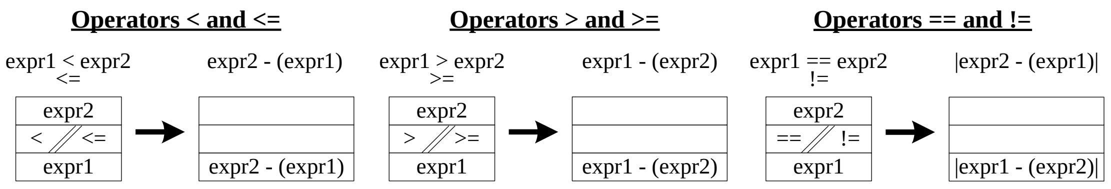


Figure 3. Transformation of relational expressions.

The syntactic tree (Figure 4) shows the evaluation order for the expressions, which is determined by the grammar of the expressions stack, where the AND operator has priority over OR operator. The instrumentation is performed following this evaluation order. In the first place, the instrumenter generates code for "*c && d*", then for "*e && f*" and finally for the OR expression.

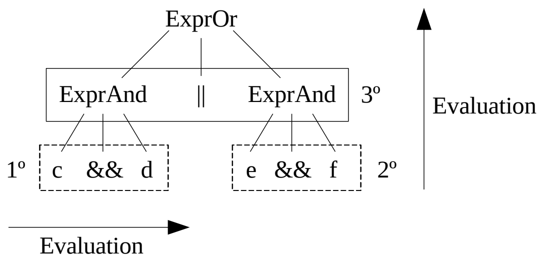


Figure 4. Syntactic tree.

When the instrumenter finds a NOT operator in the expressions stack, this operator is propagated by the grammar, until arriving at terminal expressions, causing an AND operator to become an OR operator and vice versa. In order to realize this treatment, the priorities of the operators are inverted, i.e. the OR will have precedence over the AND.

2.3 Test cases generator

This module is in charge of generating test cases, using some of the existing techniques, for example, random, tabu search [6], [7] and genetic algorithms [8]. The results presented in this paper have been obtained using two different test cases generator: one based on tabu search and another based on random technique.

3. Results

In this section, we study the efficiency of our tool, using a famous benchmark in software testing: the classify triangle program for real input variables with 3 digits of real precision. We compare the total time needed to carry out the test using a manual instrumentation and using an automated instrumentation. This total time consist of the instrumentation time and the run time.

We execute the benchmark with a manual instrumentation and an automated instrumentation for the random technique and the tabu search technique. Figures 5 and 6 show the results. In these Figures, the horizontal axis represents the time in seconds (it is showed in a logarithmic 10 scale) and the vertical axis the % accumulated branch coverage.

When the instrumentation is by hand, as it is seen in the figure 5, the needed time to carry out the instrumentation of the program under test is much greater when a no-random technique is used. On the other hand, the random testing is faster than tabu search until it achieves about 65% branch coverage, but from this point, the tabu search obtains better results, reaching furthermore, 100% coverage. Therefore the tabu search achieves better coverage results and its run time is less than random technique.

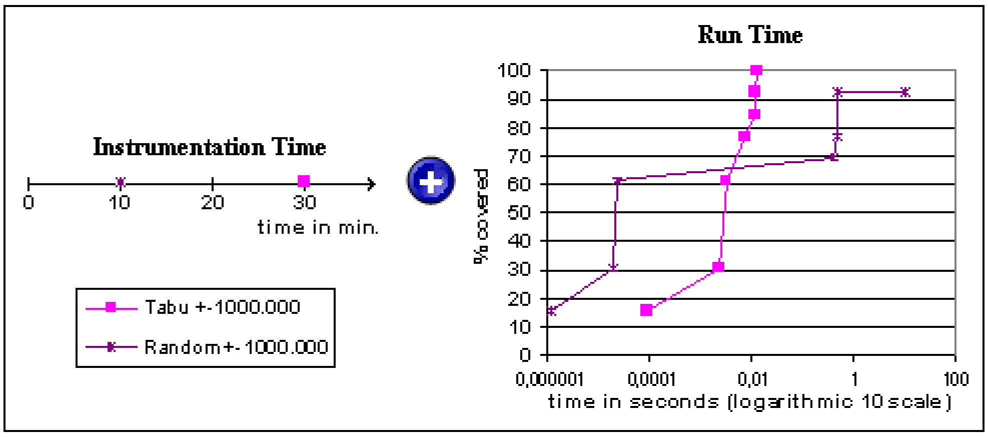


Figure 5. % accumulated coverage vs. total time for the Triangle problem (manual instrumentation)

The manual instrumentation time increases exponentially to the number of branches of the program under test. So, more complex programs have more instrumentation cost.

When the instrumentation is automatic, as it is seen in the figure 6, again the tabu search achieve 100% coverage, whereas the random technique doesn't reach it. Besides, the obtained times (instrumentation + run) for both techniques are very similar among themselves, due to the elimination of the cost to perform the manual instrumentation. Moreover, these times are much less than the total times obtained with the manual instrumentation.

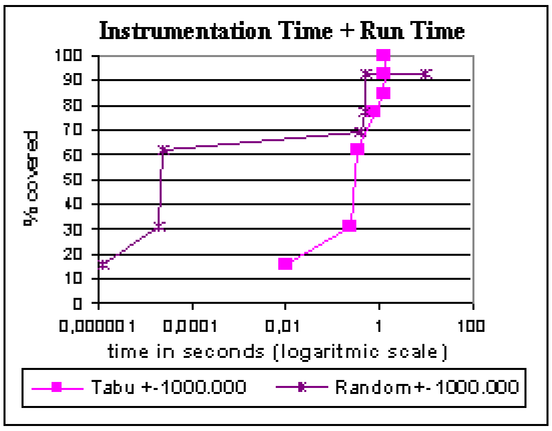


Figure 6. % accumulated coverage vs. total time for the Triangle problem (automatic instrumentation)

In short, our tool allows to decrease the testing time and furthermore another very important aspect, it eliminates the introduction of errors due to the manual instrumentation.

4. Conclusions and further work

Our tool allows to instrument automatically a program written in C/C++ and to generate automatically branch coverage software test cases.

When a program is tested, the needed time consists of instrumentation time and running time. If the instrumentation is by hand, the first one is much greater than the second one, and therefore it interests to decrease it by means of its automation.

With our modular tool we have achieved to decrease the total time needed to carry out coverage in software testing. Besides, the introduction of errors in the instrumentation is avoided due to the automation of this process.

The modularity of the tool allows to use automatically different techniques for the generation of coverage software test cases.

We are currently working on improving the tool by means of using another type of software coverage (path coverage, multiple condition coverage and loop coverage).

Acknowledgements

This work was funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, project TIC2001-1143-C03-03

References

[1] Beizer, B. Software Testing Techniques. 2nd. Ed. Van Nostrand Reinhold. 1990
[2] Cantata: http://www.iplbath.com
[3] Chang, K., Cross, J., Carlisle, W., Liao, S. A performance evaluation of heuristics_based test case generation methods for software branch coverage. International Journal of Software Engineering and Knowledge Engineering, 6(4):585-608. 1996
[4] DeMillo, R. A., Offutt, A. J. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 17(9):900-910. 1991.
[5] Gallagher, M. J., Narasimhan, V. L. Adtest: a test data generation suite for ada software systems. IEEE TSE, 23(8): 473-484. 1997
[6] Glover, F. Tabu search part i. ORSA Journal on Computing, 1(3):190-206. 1989.
[7] Glover, F. Tabu search part ii. ORSA Journal on Computing, 2(1):4-32. 1990.
[8] Goldberg, D. Genetic Algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA.1989.
[9] Korel, B. Automated software test data generation, IEEE TSE, 16(8): 870-879. 1990.
[10] Meudec C. ATGen: automatic test data generation using constraint logic programming and symbolic execution. Journal of Software Testing, Verification and Reliability, 11(2):81-96. 2001
[11] Michael, CC., McGraw, G., Schatz MA. Generating software test data by evolution. IEEE TSE, 27(12):1085-1110. 2001
[12] Ntafos, S. On random and partition testing, Intl. Symp. on Software Testing and Analysis. 1998
[13] Rational: http://www.rational.com
[14] Wegener, J., Baresel, A., Sthamer, H. Evolutionary test environment for automatic structural testing. Information & Software Technology, 43(14): 841-854. 2001