LDRA

Software Technology

Testing Times for Real Time Software

LDRA looks at ways to reduce the cost of testing and maintaining code

Article by Bill StClair originally published in Automotive Electronics Magazine February/March 2005

automotive electronics

Testing Times for Real Time Software

Testing and maintenance of code constitute 70% or more of the typical software lifecycle, but what strategies can be used to manage these costs? Bill StClair at LDRA explains.

For many years Coverage Analysis techniques have given the avionics industry consistent, cost-effective error detection when applied to the analysis of complex software control systems. Now, as the automotive industry moves towards similar reliance upon software control with similar concerns of testing quality versus spiraling costs, it would seem prudent to take note of the important lessons from the avionics industry.

There are many types and levels of Coverage. Often the term is applied to what is Function Coverage, a measure that reports whether you invoked each function or procedure. It is useful during preliminary testing to assure at least some coverage in all areas of the software and to eliminate gross deficiencies in a test suite quickly.

In companies producing complex real time software (RTS) this level of Coverage is generally not sufficient and therefore, any Coverage technique that is applied is typically expanded to include Statement and Branch Coverage.

Statement Coverage – also called line coverage, segment coverage or basic block coverage - reports whether each executable statement is touched, while basic block coverage is the same as statement coverage except the unit of code measured is each sequence of non-branching statements. This greatly expands the range of the Coverage graph, though only on a two-dimensional plane.

Statement Coverage is the easiest of coverage metrics to maximise. It covers the whole of the source code and helps the user to detect many defects that may reside within infrequently used areas. As it is relatively easy to maximise, it is not very expensive or resource consuming, yet still improves confidence in the correctness of the source code to a great degree. However, by definition, simply exercising at 100% Statement Coverage means that there are potentially paths through those statements that have not yet been explored, and Statement Coverage cannot measure them.

Statement Coverage doesn't discern various control structures. For example, consider the following C/C++ code fragment:

```
int* p = NULL;
if (condition)
   p = &variable;
*p = 123;
```

Without a test case that causes condition to evaluate false, Statement Coverage rates this code fully covered. In fact, if condition ever evaluates false, this code fails. This is the most serious shortcoming of Statement Coverage. If-statements are very common.

> *"Statement Coverage means that there are potentially paths through those statements that have not yet been explored"*

Statement Coverage also does not report whether loops reach their termination condition - only whether the loop body was executed. Since do-while loops always execute at least once, Statement Coverage considers them the same rank as non-branching statements.

Statement Coverage also ignores the logical operators (|| and &&). Moreover, Statement Coverage cannot distinguish consecutive switch labels.
In order to have completeness and accuracy, at least in terms of this two-dimensional Coverage graph, obviously Coverage would need to factor the evaluation of Decisions and the resulting code branches.

```
if (condition1 && (condition2 || function1()))
   statement1;
else
   statement2;
```

But for many embedded applications, as suggested by this example, Coverage must be expanded to evaluate conditions and by inference, the state of variables. Referred to as Condition Coverage, it reports the true or false outcome of each Boolean sub-expression, separated by logical-and and logical-or if they occur. Condition Coverage measures the sub-expressions.

"Evidence suggests that Test Path Coverage is the most effective coverage technique for maximising software quality and reliability"

In all but safety-critical applications, analysing individual conditions as discrete events can lead to overlooking the opportunity that a Coverage metric which effectively subsumes all other Coverage measures offers. Consider the gains in error prevention that can be achieved if real test scenarios are envisioned in a "white box" testing context.

How to Achieve Sufficient Coverage

How do you begin to get to a thorough level of Coverage while achieving your productivity goals such as Time to Market and Time to Quality?

Statement Coverage and Branch Coverage can normally be made to reach unity without great effort (although infeasible branches and code may be discovered), but Test Path Coverage often lags Statement and Branch by some margin, because maximising this requires a demanding testing strategy.

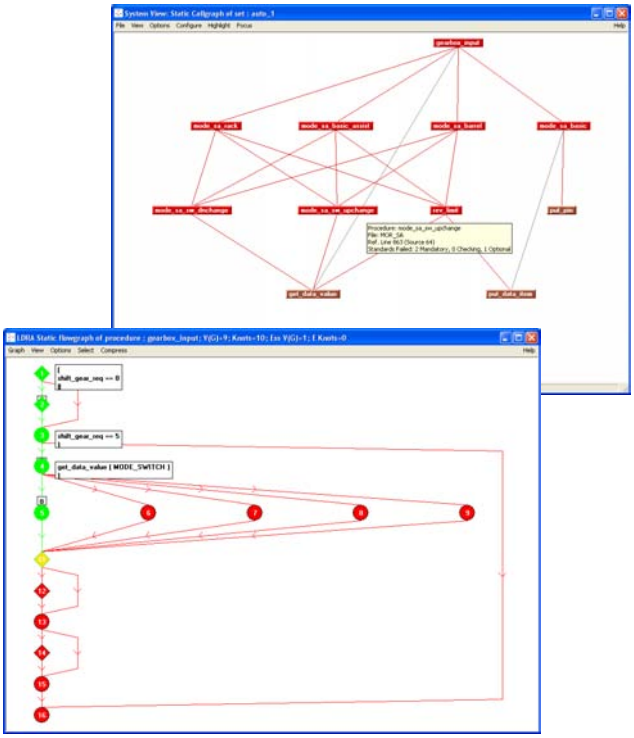
If unity can be achieved for Test Path Coverage, then the number of undetected errors remaining in the subject program is substantially reduced. Maximising Test Path Coverage is a very thorough test of a program, and is especially good at detecting errors in looping constructs. Covering every statement does not require loops to be covered at all - a straight through path is all that is required. Testing every branch ensures that a loop is executed once, but testing every path however, also requires every loop to be covered at least twice. For high integrity code, it is recommended that Test Path Coverage be maximised.

Evidence suggests that Test Path Coverage is the most effective coverage technique for maximising software quality and reliability. The goal is to maximise this and other associated coverage metrics at the minimum effort and cost.

Increasingly software developers are turning to commercially available software test tools to assist with the complex test process that such techniques demand. The more sophisticated of these tools also incorporate rules based analysis options and other static analysis techniques together with facilities for applying these powerful analysis techniques at the unit (single function) level up to sub-system and system (multi-file).

So what does your Coverage tool need to provide?

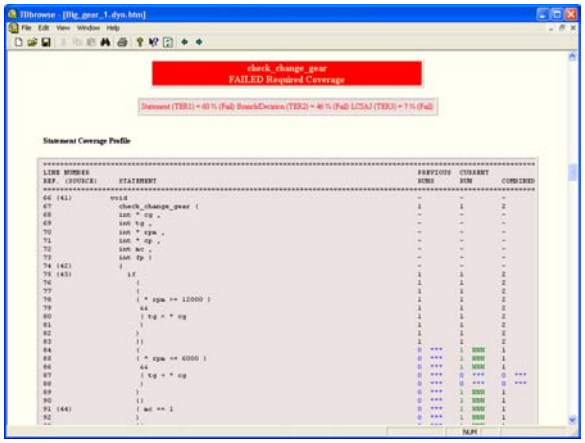
1. Exercise Coverage strategically (identifying infeasible and unreachable code as well)
2. Facilitate achievement of Coverage using automated test case/vector generation from the unit level upwards
3. Provide Visualisation and intuitive presentation of Coverage results for engineering analysis and guidance



Coupled with the use of tools developers should also consider the following steps to a more efficient, cost-effective means of achieving their Coverage analysis goals:

STEP 1: Construct the best possible functional tests from a knowledge of what the software is supposed to do. The source of this information should be a requirements specification, a program specification or user documentation. The execution of the source code with this test data should then be monitored with the aid of a test tool. When ideas for functional test data are exhausted, inspection of the Coverage data will indicate those areas of the program which have not yet been adequately tested. Further test data sets should then be constructed and their execution analysed. The Coverage accumulates the results from each test data set and notes which parts of the program were executed by each test data set.

This process is continued until either ideas for functional test data are exhausted or the required test metrics are satisfied. If the former is true proceed to Step 2; otherwise the task is completed.



STEP 2: Examine the test coverage metrics. If Statement coverage is not unity (i.e. every statement has not been executed) it is probably due to a failure to test special cases, error exits, etc. Because of these possibilities, it is essential to accumulate the execution history profile because it is usually necessary to run the program a number of times to execute every line of code. It is often found that the functional tests cover only 40-60 per cent of the executable statements.

When Statement coverage attains unity and every statement has been executed, it is then time to move to Step 3.

STEP 3: Examine any unexecuted branches. Some of these branches can usually be executed by constructing special cases. When this strategy is exhausted it is more cost-effective to move on to Step 4 - Test Path Coverage since the program analysis needed to explore the remaining unexecuted branches is similar to that needed for the unexecuted Test Paths.

STEP 4: Some unexecuted branches and Test Paths may be traced to causes such as special cases which can arise only under error states, either of the program or of underlying computational processes. This is often referred to as defensive programming and these Test Paths should be left intact.

Finally

It will often be found on further inspection that many of the unexecuted Test Paths are infeasible, i.e. they cannot be executed for any test data. This may suggest that a portion of code needs rewriting because it is either inelegant, inefficient or incorrect. Furthermore, when this code is rewritten, other Test Paths which were related to the poor code may also have been removed. Some infeasible Test Paths will be considered inoffensive and can be left at the price that program readability is reduced. Provided the cause is known, these Test Paths may be ignored with unity unattainable. If the infeasible Test Paths are removed then the source code will be more efficient, robust and occupy less space.

Bill StClair is a Technical Evangelist at LDRA

LDRA Headquarters
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Inc. (US)
Lake Amir Office Park
1250 Bayhill Drive Suite # 360
San Bruno CA 94066
Tel: (650) 583 8880

www.ldra.com