Make Sense of Software of Unknown Pedigree (SOUP)

Mark Pitchford, LDRA, www.LDRA.com

Software test tools have been traditionally designed with the expectation that the code has been (or is being) designed and developed following an ideal development process which adheres to a best practise approach. Such an approach implies the existence of clearly defined requirements, an adherence to corporate or international coding standards, a well-controlled development process, and a coherent test regime.

Legacy code turns the ideal process on its head. Although such code is a valuable asset, proven in the field over many years, it was likely to have been developed on an experimental, ad hoc basis by a series of “gurus” – experts who prided themselves at getting things done and in knowing the application itself, but not necessarily expert at complying with modern development thinking and bored with providing complete documentation.

Frequently, this legacy software—software of unknown pedigree (SOUP)—forms the basis of new developments which must meet modern coding standards either due to client demands or simply a policy of continuous improvement within the developer organization. The need to leverage the value of SOUP while meeting new standards and further developing functionality presents its own set of unique challenges.

The Dangers of SOUP

Many SOUP projects will have initially been subjected to functional system testing which usually offers very poor code coverage, leaving many code paths un-exercised. When that code is applied in service, the different data and circumstances are likely to use many such paths for the first time and hence unexpected results may occur (Figure 1).

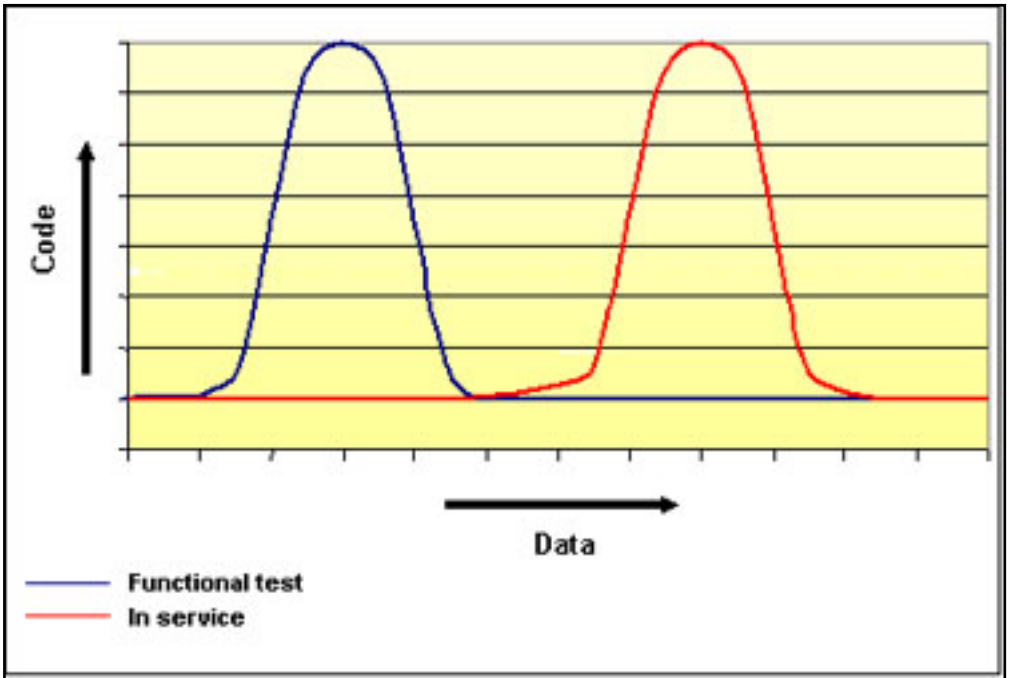

Figure 1. Traditional functional testing can leave many parts of the code unproven. The blue line shows the parts of code exercised with traditional testing while the red line demonstrates the parts of code that are used when the application runs in the field.

The European System and Software Initiative “Performance of Errors through experience-driven Test” (PET) investigated this phenomenon and agreed that deployed code often possesses many errors. Their findings demonstrated that the number of bugs that can be found by newer methods of testing, such as static and dynamic analysis, is quite large, even in code that has been through functional system testing and has subsequently been released.

It is often argued that legacy code which forms the basis of new developments has been adequately tested just by being deployed in the field. However, even in the field, it is highly likely that the circumstances required to exercise some parts of the code have never (and possibly can never) occur. It follows that many un-exercised paths are likely to remain in software tested only through functional testing and in the field, and such applications have therefore sustained little more than an extension of functional system testing by their in-field use.

When there is a requirement for ongoing development of legacy code for later revisions or new applications, previously un-exercised code paths are likely to be called into use by combinations of data never previously encountered (Figure 2).

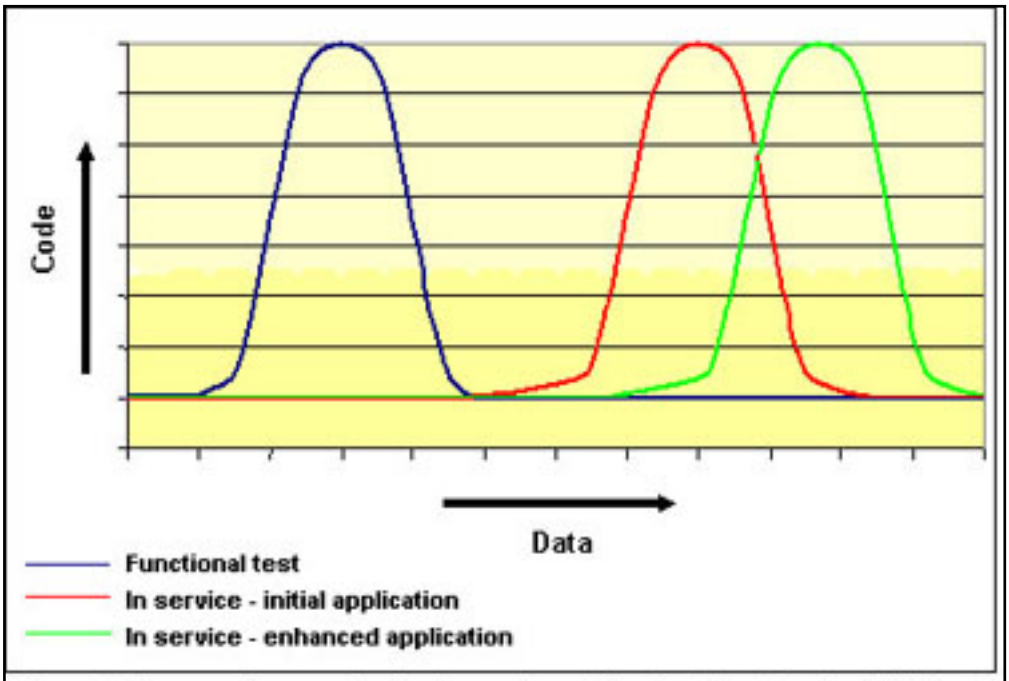

Figure 2. Even code exercised both on site and by functional testing is likely to include many unproven execution paths. Code enhancements are prone to call into service cominations of data previously un-encountered, resulting in the possibility of tapping into those previously un-exercised paths.

Such situations may become particularly challenging when the legacy code has been developed by individuals who have long since left the development team, particularly if the legacy code originates from a time when documentation was not of the high standard expected today.

Given that commercial pressures often rule out a rewrite, what options are available?

Static Analysis of SOUP's Dynamic Behaviour

Exercising all routes through legacy code may appear to be a daunting task, and there are a number of static analysis tools on the market which use mathematical techniques such as abstract interpretation to try to verify all possible executions of a program. As a result, they aim to prove which operations are free of run-time errors, including overflows, divisions by zero, buffer overflows, or pointer issues, and which identify where run-time errors will or might occur.
Such claims sound appealing, especially if engineers believe the half truth that problems can be isolated without any need for code to be changed or even understood.

It is indeed true that some problems, in simpler code sections can be readily isolated and corrected with relatively little knowledge of the code, and that such successes can provide great encouragement during tool evaluation. It is easy to derive some level of warm, fuzzy feelings that the software is reasonably robust in this way.

However, where source code is complex such tools have to rely ever more heavily on data approximations and hence raise many false warnings. These require the

user to confirm or deny the existence of each problem in the very code sections which by definition are likely to represent the most complex parts of the application and the most obtuse parts of the sources.

Even setting that aside, these tools provide no evidence that the code is functionally correct – and software which is proven to be robust but which remains functionally flawed is unusable.

Using Static and Dynamic Analysis in Combination

If the panacea of purely static techniques is a mirage, then how can the more traditional combination of static and dynamic methodologies be adapted to address the unique demands of SOUP?

The traditional application of formal test tools demands the sequence outlined in Figure 3.

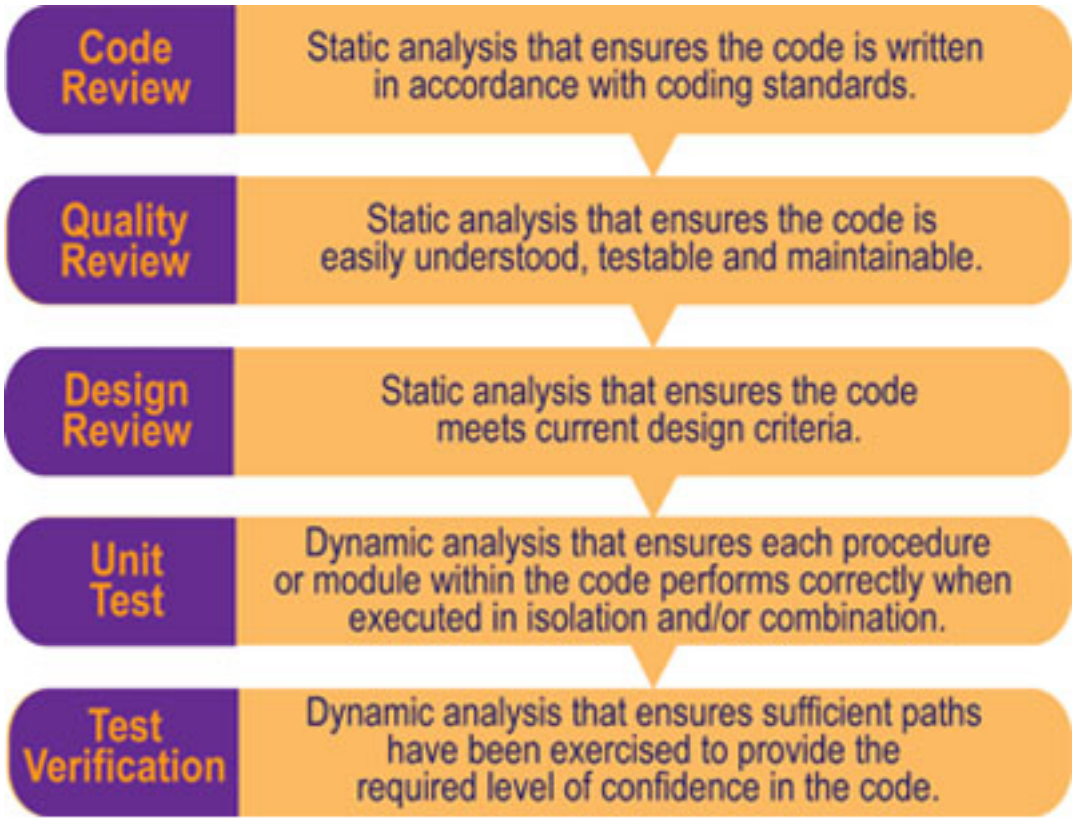

Figure 3. Traditional sequence for the application of test tools. Note that each step of the process undergoes testing designed to ensure the code is error free, exercised, and complies with application requirements.

The team facing the task of building on SOUP will be required to follow a more pragmatic approach not only because the code already exists, but also because the code itself may well be the only available detailed (or at least current) “design document”. In other words, the existing code effectively defines the functionality of the system rather than any documentation. In enhancing the existing code, it is vital that the existing functionality is not unintentionally modified either by rewriting to meet coding

standards or as the result of the enhancements under development.

The challenge is therefore to identify and use the building blocks within the test tools which, when used in a different sequence, can help in creating an efficient enhancement of SOUP.

Enhancing SOUP with a Modern Test Tool Suite

To decide on a more pragmatic approach, it is necessary to answer some fundamental questions:

1. What level of understanding of the existing code is available to the development team?
2. Does the existing code need to be upgraded to meet any new standards not previous enforced?
3. How can the sections of code affected by changes be tested? Can code coverage levels be proven to have adequately exercised that code?
4. What if the nature of the code is such that modularity is compromised? How can new sections of code then be proven?
5. How can the revised code be proven to be functionally equivalent to its SOUP origins?

Improving the Level of Understanding

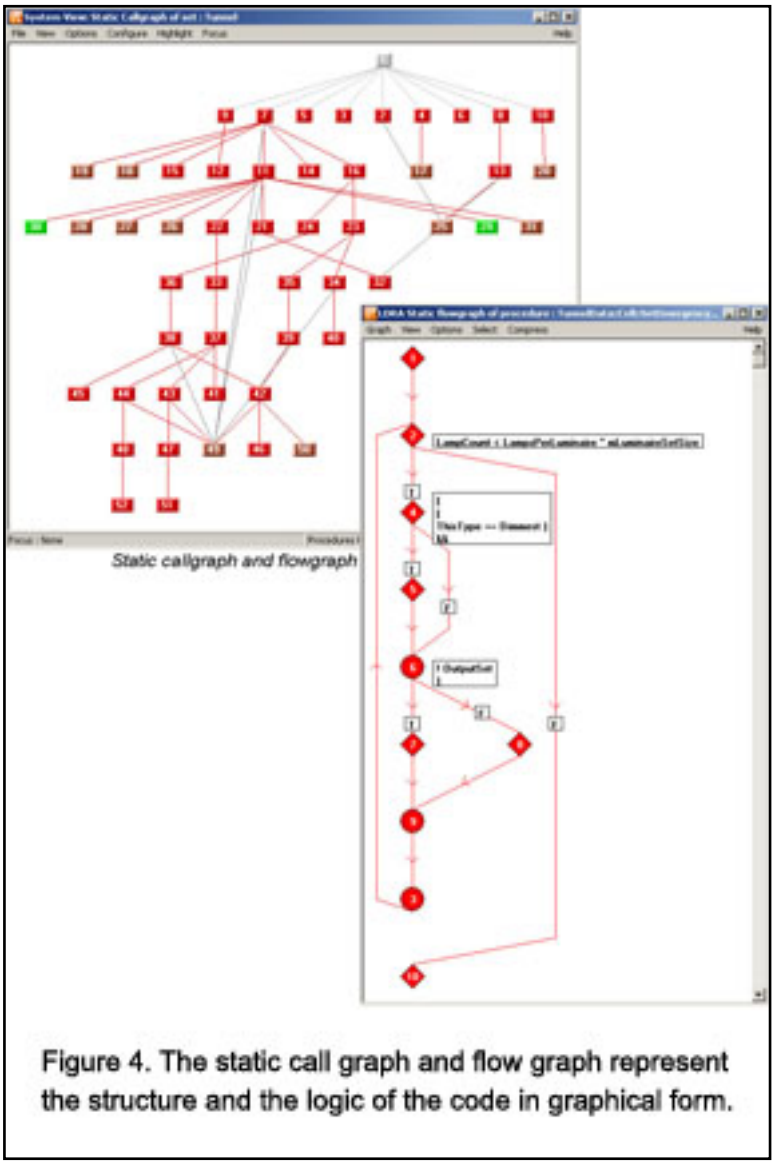

Figure 4. The static call graph and flow graph represent the structure and the logic of the code in graphical form.

The system visualisation facilities provided by modern test tools is extremely powerful, whether applied in terms of statement blocks, procedures (or classes), applications and/or system wide. Static call graphs provide a hierarchical illustration of the application and system entities, and static flow graphs show the control flow across program blocks, as shown in Figure 4. The use of such colour-coded diagrams can be of considerable benefit when understanding SOUP.

Such call graphs and flow graphs are just part of the benefit of the comprehensive analysis of all parameters and data objects used in the code. They are complemented by utilities such as automatic header comment generation, and data flow reporting to provide details of the relationships between the component parts of the software and problems associated with those relationships. Such information is particularly vital to enable the affected procedures and data structures to be isolated and understood when work begins on enhancing functionality.

Enforcing New Standards

When new developments are based on existing SOUP, it is likely that internal, industry, or international standards will have been enhanced in the intervening period. Code review analysis can highlight any contravening code for correction. It may be that the enforcement of a full set of current coding rules to SOUP is too onerous and so a subset compromise is preferred. In that case, it is possible to apply a user-defined set of rules as illustrated in Figure 5, which allows such a subset to retain cross referencing to the international standard.

Where legacy code is subject to continuous development, it is likely that production pressures generally outweigh the longer term ideal to adhere to more demanding standards. It may be pragmatic to initially establish a relatively lenient set of coding rules, to isolate only the most unwanted violations. A progressive transition to a higher ideal may then be made by periodically adding more rules with each new release, so that the impact on incremental functionality improvements is kept to a minimum.
Test tools cannot correct the code. However, they enable the correction of code to adhere to such rules as efficiently as possible. Using a “drill down” approach, the test tools provide a direct link between the description of a violation in a Code Review Report and an editor opened to the line of code which includes that violation.

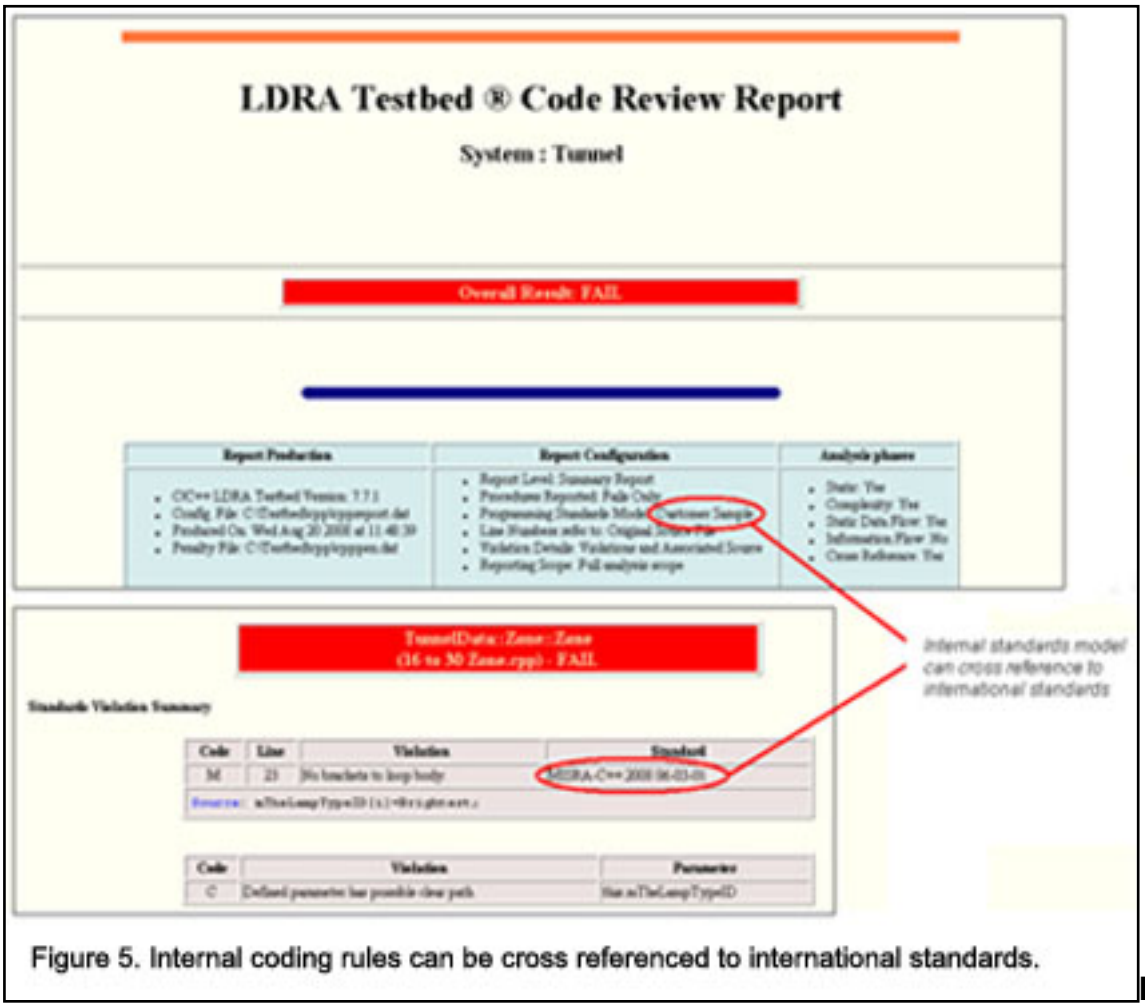

Figure 5. Internal coding rules can be cross referenced to international standards.

Ensuring Adequate Code Coverage

As established earlier, testing based on a statistical sample of a program's expected input data can lead to testing being focused on a particular value or range of values. Code proven in service has effectively been subjected only to similar, if extensive, functional testing which has serious implications for SOUP enhancement. Structural Coverage addresses this issue by testing equally across the data, assuming each value has an equal chance of occurring.

System Wide Testing

Although system-wide functional testing will not test all paths, clearly it will test many. Given that functional software exists at the outset, it provides a logical place to start. Test tools can provide the means to identify which parts of the software have been exercised, and so highlight those areas still requiring attention.
The test tool takes a copy of the code under test and implants additional function calls ("instrumentation") to identify the paths exercised during execution. Using this "white box" testing technique, tools provide statistics to show how much of the code has been used. Coloured call graphs and flow graphs complement reports to give an insight into the code tested, and to clearly show the nature of data required to ensure additional coverage.

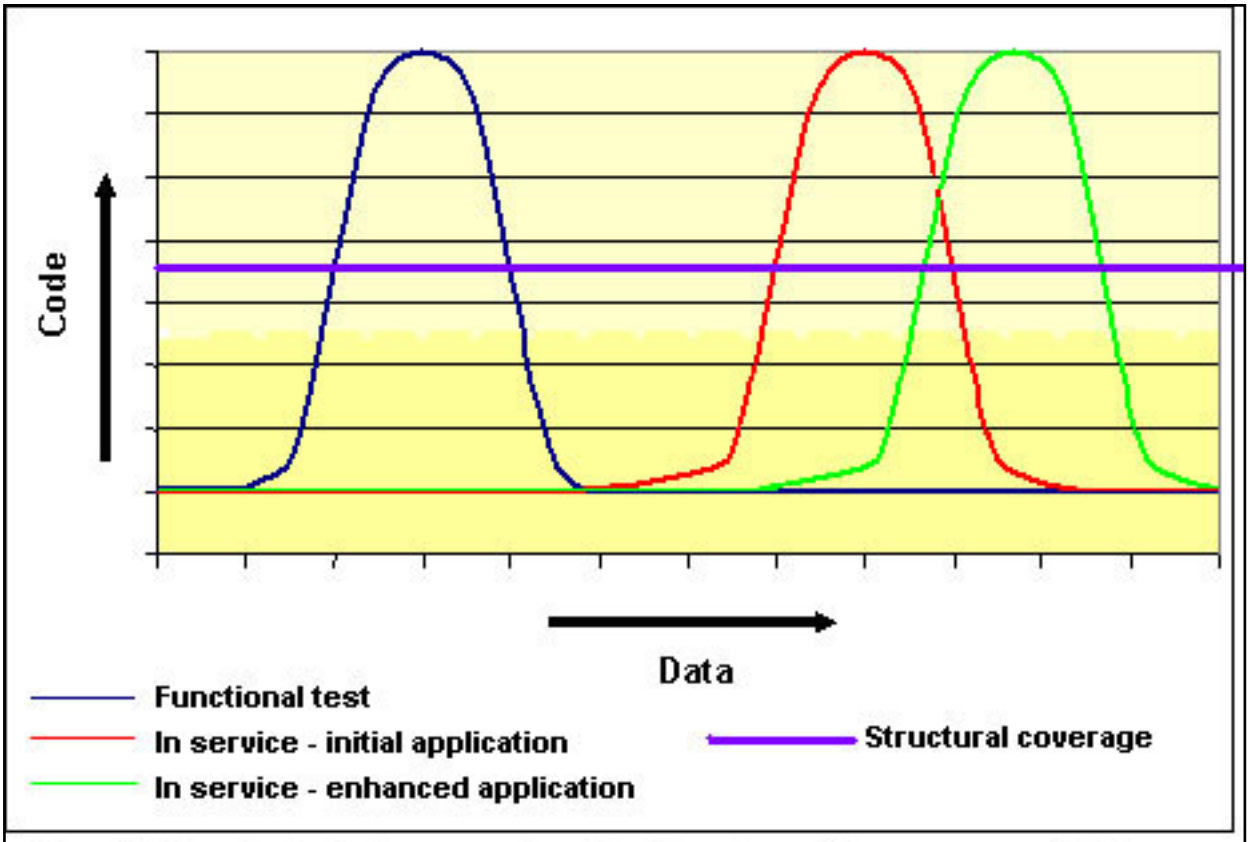

Figure 6. Adequate structural coverage throughout the code provides reassurance that changes in data or functionality will not expose untested code paths.

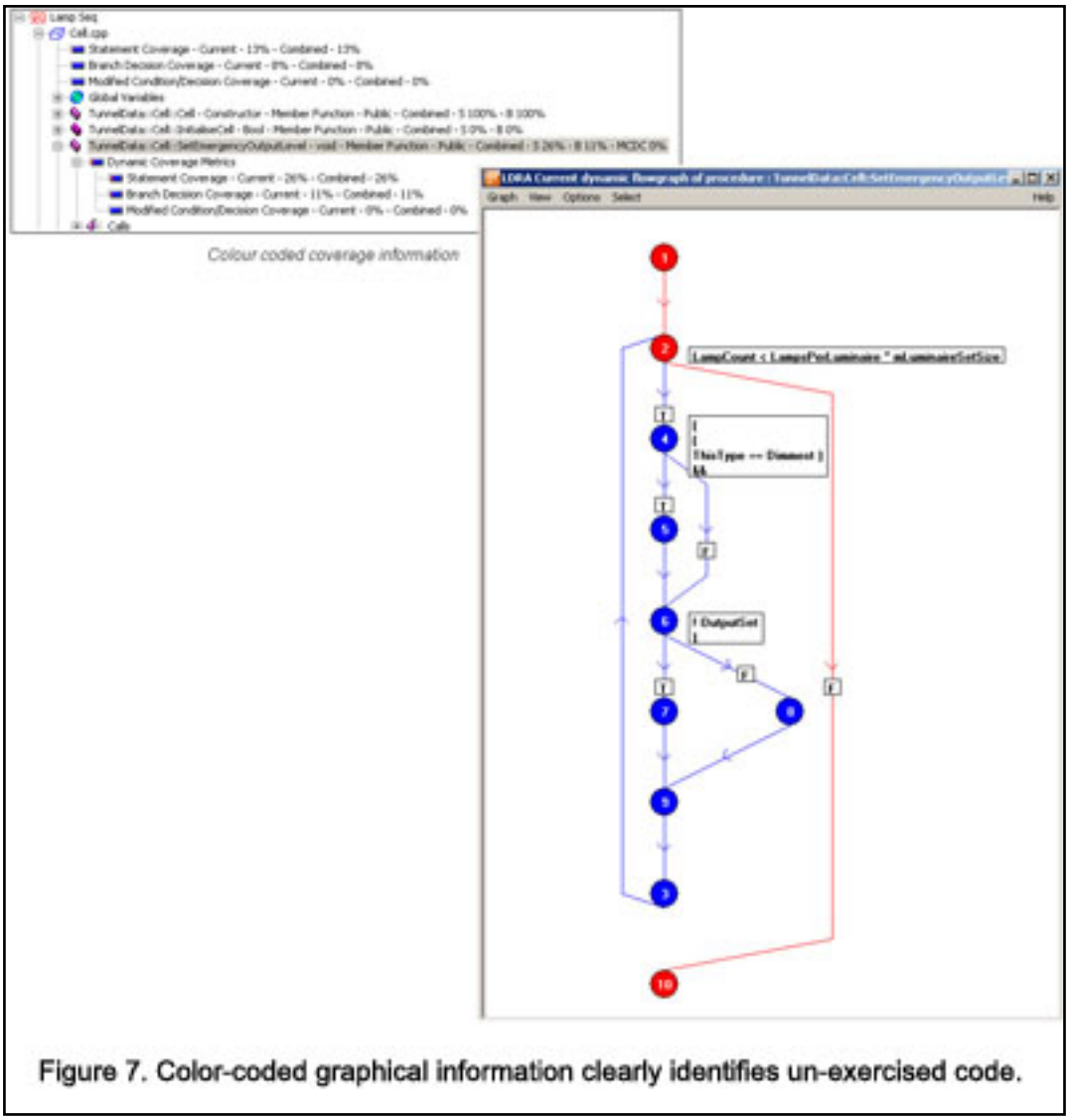

Figure 7. Color-coded graphical information clearly identifies un-exercised code.

Unit Testing

System wide testing inevitably leaves some paths unproven. However, because code instrumentation can be applied on a unit test or system wide basis, it is possible to devise unit tests to exercise those parts of the code which have yet to be proven through system test. This is equally true of code which is inaccessible under normal circumstances, such as exception handlers and defensive coding. Unit testing can be used to ensure that each section (or unit) of the code functions correctly in isolation. Whether testing a single function/procedure or a subsystem within an application, the time involved in manually constructing a harness to allow the code to compile can be considerable and can demand a high level of expertise on the part of the tester.

Modern unit test tools minimize that overhead by automatically constructing the harness code within an easy-to-use GUI environment and providing details of the input and output data variables to which the user may assign values. Such tools can overcome traditional headaches such as providing access to C++ private member variables for test purposes. The result can then be exercised on either the host or target machine.

Sequences of these test cases can be stored, and they can be automatically exercised regularly (perhaps overnight) to ensure that ongoing development does not adversely affect proven functionality.

Dealing with Compromised Modularity

In some SOUP applications, terminology such as “Unit test” and “subsystem” do not readily spring to mind. Structure and modularity often suffer in such code bases, which does not make them any less vital to the organization but can challenge the notion of testing functional or structural subsections of that code.

However, unit test tools can be very flexible in these matters, and the harness code which is constructed to drive test cases can include as much or as little of the source code base as is deemed necessary by the user. If it is necessary to include a large section of it to test the behavior of particular functions within a system, then so be it.

The ability to do that may be sufficient to suit a purpose: “letting sleeping dogs lie.” However, if a longer term goal exists to improve matters, then using instrumented

code can be a vital tool in helping to understand which execution paths are taken when different input parameters are passed into a function – either in isolation or in the broader context of its calling tree.

Ensuring Correct Functionality

From a pragmatic viewpoint, perhaps the most important aspect of SOUP-based development is ensuring that all aspects of the software functions as expected, despite changes to the code and to the data handled by it. In making changes to the code either to meet standards or to add functionality, it is therefore of paramount importance that functionality is not inadvertently changed.

Of course, unit tests could be used throughout to ensure that is the case but even with the aid of test tools, that may involve more work than the budget will accommodate. However, the concern here is not checking that each function is working in a particular way; more that however it works beforehand, it is not changing, except when there are deliberate attempts to make it do so.

Automatic test case generation is key here. By statically analyzing the code, test tools can automatically generate test cases to exercise a high percentage of the paths through it. Input and output data to exercise the functions is generated automatically, and then retained for future use.

They can then be used to perform batch regression testing on an ongoing basis, perhaps overnight or weekly, to ensure that when those same tests are run on the code under development there are no unexpected changes recorded. These regression tests automatically provide the cross reference back to the functionality of the original source code.

This ensures that when the enhanced software is released, clients can benefit from the new functionality and improved robustness of the revised code without fear of nasty surprises!

Conclusion

In an ideal world, test tools should be applied from the beginning of a structured and formal development process. However, sometimes commercial realities mean that legacy SOUP code is used as a basis for further development. By using those same test tools in a pragmatic way in these circumstances, it is possible to develop legacy SOUP code into a sound set of sources which are proven to be fit for purpose – all in an efficient and cost effective manner.

About the Author

Mark Pitchford has over 25 years’ experience in software development for engineering applications, the majority of which have involved the extension of existing code bases. He has worked on many significant industrial and commercial projects in development and management, both in the UK and internationally including extended periods in Canada and Australia. For the past 7 years, he has specialised in software test and works throughout Europe and beyond as a Field

Applications Engineer with LDRA.