9

EMBEDDED SOFTWARE QUALITY, INTEGRATION, AND TESTING TECHNIQUES

Mark Pitchford
LDRA, Monks Ferry, United Kingdom

CHAPTER OUTLINE

Software Engineering for Embedded Systems. https://doi.org/10.1016/B978-0-12-809448-8.00009-6

1 What Is Software Test?

There is some inconsistency in how the word "test" is used in the context of software development. For some commentators "software test" implies the execution of software and the resulting confirmation that it performs as was intended by the development team—or not. Such a definition views the inspection or analysis of source code as a different field; that is, one to be contrasted with software test rather than a branch of it.

For the purposes of this chapter the *Oxford English Dictionary*'s definition of the word "test" is applied: "a procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use."

Any activity which fits that definition can therefore be regarded as a software test, whether it involves code execution or not.

The generic term "static analysis" is used to describe a branch of software test involving the analysis of software without the execution of the code. Conversely, "dynamic analysis" describes a branch of software test in which the code is indeed executed.

2 Why Should We Test Software?

Returning to the definition of "test," software is tested to establish its "quality, performance, or reliability." Testing itself only establishes these characteristics; it does not of itself guarantee that software meets any particular criteria for them.

The aim then is to quantify the standard of the software. Whether that standard is good enough depends very largely on the context in which it will be deployed.

3 How Much Testing Is Enough?

One approach to static analysis focuses on the checking for adherence to coding rules or the achievement of particular quality metrics. Such an approach is usually easy enough to scope. Either code meets the rules or it does not, and if it does not it is either justified or corrected.

Other static analysis tools are designed to predict the dynamic behavior of source code. Such heuristic mechanisms are most commonly

used for "bug finding"—looking for evidence of where source code is likely to fail, rather than enforcing standards to be sure that it will not. These are arguably complementary to other static and dynamic techniques and are often easy to apply to get a marked improvement in code quality. However, they sometimes lack the thoroughness sought in the development of mission-, safety-, or security-critical software.

Dynamic testing is less easy to apply in even fairly trivial applications. The possible combinations and permutations of data values and execution paths can be large enough to make it wholly impractical to prove that all possible scenarios are correctly handled.

This means that almost irrespective of how much time is spent performing software tests of whatever nature an element of risk will remain with regard to the potential failure of those scenarios that remain unproven.

Consequently, the decision on what and how much to test becomes a question of cost vs. the impact of the risk outcomes identified. Those risk outcomes include not only the risk of software failure, but also factors such as the risk of delaying the launch of a commercial product and conceding the initiative in the market to a competitor.

Testing is not a cheap activity and there is the cost of both labor and associated test tools to take into account. On the opposite side of the equation lies the consequence of flawed software. What is the likely outcome of failure? Could it kill, maim, or cause temporary discomfort? Could it yield control of the application to bad actors or make personally identifiable information (PII) accessible? Or is a mildly irritating occasional need to restart the application the only feasible problem?

Clearly, the level of acceptable risk to health, safety, and security in each of these scenarios is significantly different, and the analysis is further complicated if there are also commercial risk factors to be added into that equation.

Some standards such as IEC 61508 (see Section 7) define a structured approach to this assessment. In this standard, software integrity level (SIL) 1 is assigned to any parts of a system in continuous use for which a probability of failure on demand of 10^{-5}–10^{-6} is permissible. SILs become more demanding the higher the number assigned, so that SIL2 implies an acceptable probability of failure on demand as 10^{-6}–10^{-7}, SIL 3 as 10^{-7}–10^{-8}, and SIL 4 as 10^{-8}–10^{-9}.

The standard recommends the application of many techniques to varying degrees for each of these SILs on the basis that the proficient application of the specified techniques will provide sufficient evidence to suggest that the maximum acceptable risk level will not be exceeded.

Ultimately, then, the decision is about how the software can be proven to be of adequate quality. In many cases this ethos allows different SIL levels to be applied to different elements of a project depending on the criticality of each such element.

That principle can, of course, be extended outside the realms of high-integrity applications. It always makes sense to apply more rigorous test to the most critical parts of an application.

4 When Should Testing Take Place?

To some extent that depends on the starting point. If there is a suite of legacy code to deal with, then clearly starting with a new test regime at the beginning of development is not an option! However, "the sooner, the better" is a reasonable rule of thumb.

In general terms the later a defect is found in product development, the more costly it is to fix—a concept first established in 1975 with the publication of Brooks' *The* "Mythical Man-Month" and proven many times since through various studies.

The automation of any process changes the dynamic of justification, and that is especially true of test tools given that some are able to make earlier unit test much more feasible (Fig. 1).

5 Who Makes the Decisions?

It is clear that the myriad of interrelated decisions on what, when, why, how, and how much to test is highly dependent on the reasons for doing so.

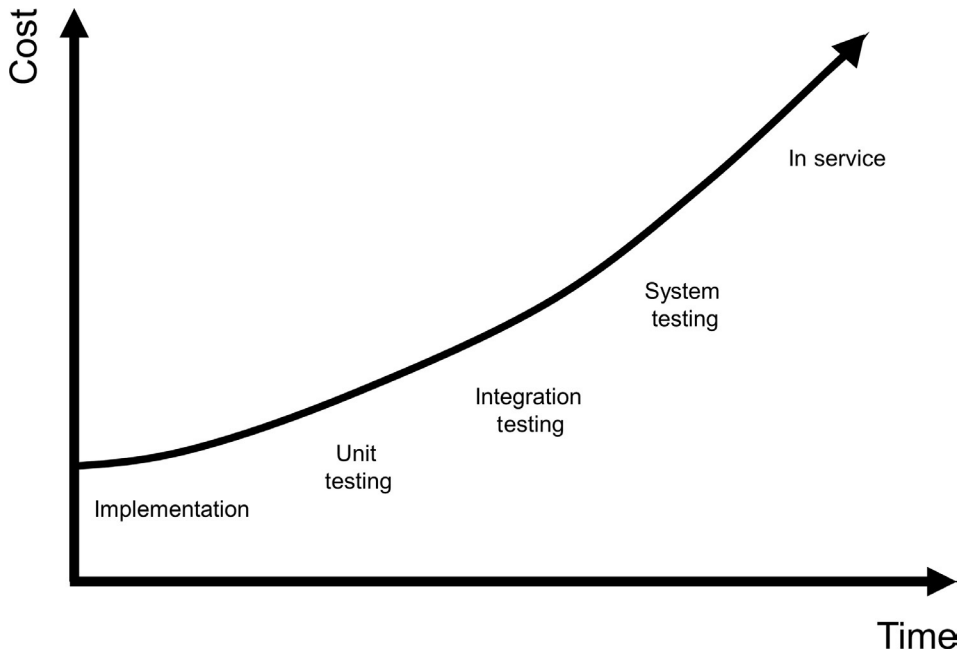


Fig. 1 The later a defect is identified, the higher the cost of rectifying it.

The judgments are perhaps relatively straightforward if an outside agency is involved. For instance, the developers of a control system for use in an aircraft will have to adhere to DO-178C for their product to become commercially available for use in international airspace. It then becomes clear that if the product is to sell, then a level of test that is appropriate to the standard is unavoidable.

This extends further to the qualification or certification of any tools to be used. That can vary quite significantly from one standard to another, but there are usually guidelines or instructions are laid down on what is required.

Conversely, the driver might be an internal one to improve software quality and improve corporate reputation and reduce recall costs. In that case the matter is a decision for management who will need to make judgments as to how much investment in the associated work and tools is appropriate.

6 Available Techniques

Enter the phrase "software test" into any browser and the variation in scope of test techniques and test tools is daunting. Static analysis, coding standards, quality metrics, source code coverage, object code coverage, dynamic analysis, memory leak profiling, abstract interpretation … the list of buzzwords and techniques is seemingly endless.

The resulting confusion is compounded by the fact that the boundaries between different techniques and approaches are not as clear-cut as they might be. "Static analysis" is a prime example of a term that means different things to different observers.

6.1 Static and Dynamic Analysis

The generic term "static analysis" is used only to indicate that analysis of the software is performed without executing the code, whereas "dynamic analysis" indicates that the code is indeed executed. So, simple peer review of source code and functional test fit the definitions of static and dynamic analysis, respectively. The boundaries become blurred when it is understood that static analysis can be used to predict dynamic behavior. As a result, it is a precision tool in some contexts and yet in others it harbors approximations.

To cut through this vague terminology it is useful to consider five key elements of analysis. These are all deployed in one form or another by analysis tools, but many can be and frequently are implemented from first principles usually in combination to provide a "tool kit" of techniques.

The first three are approaches to static analysis. Note that these attributes do not comprehensively describe the categories of static

analysis tools. Many tools include more than one of these attributes, and it is possible to approximate each of them without the use of tools at all.

6.1.1 Code Review

Code review traditionally takes the form of a peer review process to enforce coding rules to dictate coding style and naming conventions, and to restrict commands available for developers to a safe subset.

Peer review of software source code was established to achieve effective code review long before any tools automated it, and is still effective today. The key to effective peer reviews is to establish a mutually supportive environment so that the raising of nonconformities is not interpreted as negative criticism.

If manual peer review is to be adopted with such standards as the MISRA ones in mind, then a subset of the rules considered most important to the developing organization is likely to yield the best results.

Many software test tools automate this approach to provide a similar function with benefits in terms of the number and complexity of rules to be checked and in terms of speed and repeatability.

Code review does not predict dynamic behavior. However, code written in accordance with coding standards can be expected to include fewer flaws that might lead to dynamic failure, and assuring a consistent approach from individuals brings its own benefits in terms of readability and maintainability.

Code review can be applied whether the code under development is for a new project, an enhancement, or a new application using existing code. With legacy applications, automated code review is particularly strong for presenting the logic and layout of such code to establish an understanding of how it works with a view to further development. On the other hand, with a new development the analysis can begin as soon as any code is written—no need to wait for a compilable code set, let alone a complete system.

6.1.2 Theorem Proving

Theorem proving defines desired component behavior and individual runtime requirements.

The use of assertions within source code offers some of the benefits of the theorem-proving tools. Assertions placed before and after algorithms can be used to check that the data passing through them meet particular criteria or are within particular bounds.

These assertions can take the form of calls to an “assert” function as provided in languages such as C++, or the form of a user-defined mechanism to perhaps raise an error message or set a system to a safe state.

Automated theorem proof tools often use specially formatted comments (or "annotations") in the native language. These comments can be statically analyzed to confirm the code accurately reflects these definitions, which are ignored by a standard compiler. Because of these annotations verification can concentrate on verification conditions: that is, checking that when one starts under some preconditions and executes such a code fragment the postcondition will be met.

The writing of annotations can be labor intensive and so these tools tend to be limited to highly safety-critical applications where functional integrity is absolutely paramount over any financial consideration (e.g., flight control systems).

Unlike the prediction of dynamic behavior through static analysis the use of "Design by Contract" principles often in the form of specially formatted comments in high-level code can accurately formalize and validate the expected runtime behavior of source code.

Such an approach requires a formal and structured development process, one that is textbook style and has uncompromising precision. Consequently, applying the retrospective application of such an approach to legacy code would involve completely rewriting it.

6.1.3 Prediction of Dynamic Behavior Through Static Analysis

The prediction of dynamic behavior through static analysis mathematically models the high-level code to predict the probable behavior of executable code that would be generated from it. All possible execution paths through that mathematical model are then simulated, mapping the flow of logic on those paths coupled with how and where data objects are created, used, and destroyed.

The net result consists of predictions of anomalous dynamic behavior that could possibly result in vulnerabilities, execution failure, or data corruption at runtime.

Although there is no practical way of exactly performing this technique manually, the use of defensive code and bounds checking within source code offers a different approach to yielding some of the benefits. For example, many of these heuristic tools use a technique called Abstract Interpretation to derive a computable semantic interpretation of the source code. In turn, this is used to analyze possible data ranges to predict any problematic scenarios at runtime. Defensive programming can make no such predictions but assertions, say, placed before and after algorithms can be used to defend against such scenarios by checking that the data passing through them meet particular criteria or are within particular bounds—and that includes checking for the circumstances that may cause runtime errors of the type generally sought out by tools of this nature.

As before, these assertions can take the form of calls to an "assert" function as provided in languages such as C++ or the form of a

user-defined mechanism, and it is highly pragmatic to use assertions in the most difficult and complex algorithms where failures are most likely to occur.

When tools are available the static prediction of dynamic behavior works well for existing code or less rigorously developed applications. It does not rely on a formal development approach and can simply be applied to the source code as it stands, even when there is no in-depth knowledge of it. That ability makes this methodology very appealing for a development team in a fix—perhaps when timescales are short, but catastrophic and unpredictable runtime errors keep coming up during system test.

There is, however, a downside. The code itself is not executing, but instead is being used as the basis for a mathematical model. As proven by the works of Church, Gödel, and Turing in the 1930s a precise representation of the code is mathematically insoluble for all but the most trivial examples. In other words the goal of finding every defect in a nontrivial program is unreachable unless approximations are included that by definition will lead to false-positive warnings.

The complexity of the mathematical model also increases dramatically as the size of the code sample under analysis gets bigger. This is often addressed by the application of simpler mathematical modeling for larger code samples, which keeps the processing time within reasonable bounds. But, increases in the number of these "false positives," which has a significant impact on the time required to interpret results, can make this approach unusable for complex applications.

The last two of the "key attributes" concern dynamic analysis. Note that these attributes do not comprehensively describe the categories of dynamic analysis and that many tools include more than one of these attributes.

An overlap between static and dynamic analysis appears when there is a requirement to consider dynamic behavior. At that point the dynamic analysis of code that has been compiled, linked, and executed offers an alternative to the prediction of dynamic behavior through static analysis.

Dynamic analysis involves the compilation and execution of the source code either in its entirety or on a piecemeal basis. Again, while many different approaches can be included, these characteristics complete the list of the five key attributes that form the fundamental "toolbox of techniques."

6.1.4 Structural Coverage Analysis

Structural coverage analysis details which parts of compiled and linked code have been executed, often by means of code instrumentation "probes."

In its simplest form these probes can be implemented with manually inserted print statements as appropriate for the programming

language of choice. Although such an approach demands in-depth knowledge of the code under test and carries the potential for human error, it does have a place in smaller projects or when only practiced on a critical subset of an application.

A common approach is for automated test tools to automatically add probes to the high-level source code before compilation.

Adding instrumentation probes obviously changes the code under test, making it both bigger and slower. There are therefore limitations to what it can achieve and to the circumstances under which it can be used, especially when timing errors are a concern. However, within appropriate bounds it has been highly successful and in particular has made a major contribution to the sound safety record of software in commercial aircraft.

Some test tools can perform structural coverage analysis in isolation or in combination with unit, module, and/or integration testing.

6.1.5 Unit, Module, and Integration Testing

Unit, module, and integration testing (referred to collectively as unit testing hereafter) all describe an approach in which snippets of software code are compiled, linked, and built in order that test data (or "vectors") can be specified and checked against expectations.

Traditionally, unit testing involves the development of a "harness" to provide an environment where the subset of code under test can be exposed to the desired parameters for the tester to ensure that it behaves as specified. More often than not in modern development environments the application of such techniques is achieved through the use of automated or semi-automated tools. However, a manual approach can still have a place in smaller projects or when only practiced on a critical subset of an application.

Some of the leading automated unit test tools can be extended to include the automatic definition of test vectors by the unit test tool itself.

Unit testing and structural coverage analysis focus on the behavior of an executing application and so are aspects of dynamic analysis. Unit, integration, and system test use code compiled and executed in a similar environment to that being used by the application under development.

Unit testing traditionally employs a bottom-up testing strategy in which units are tested and then integrated with other test units. In the course of such testing, individual test paths can be examined by means of structural coverage analysis. There is clearly no need to have a complete code set to hand to initiate tests such as these.

Unit testing is complemented by functional testing, a form of top-down testing. Functional testing executes functional test cases,

perhaps in a simulator or in a target environment, at the system or subsystem level.

Clearly, these dynamic approaches test not only the source code, but also the compiler, linker, development environment, and potentially even target hardware. Static analysis techniques help to produce high-quality code that is less prone to error, but when it comes to proving correct functionality there is little alternative but to deploy dynamic analysis. Unit test or system test must deploy dynamic analysis to prove that the software actually does what it is meant to do.

Perhaps the most telling point with regard to the testing of dynamic behavior—whether by static or dynamic analysis—is precisely what is being tested. Intuitively, a mathematical model with inherent approximations compared with code being compiled and executed in its native target environment suggests far more room for uncertainty.

If the requirement is for a quick fix solution for some legacy code that will find most problems without involving a deep understanding of the code, then the prediction of dynamic behavior via static analysis has merit. Similarly, this approach offers quick results for completed code that is subject to occasional dynamic failure in the field.

However, if there is a need to prove not only the functionality and robustness of the code, but also provide a logical and coherent development environment and integrated and progressive development process, then it makes more sense to use dynamic unit and system testing. This approach provides proof that the code is robust and that it does what it should do in the environment where it will ultimately operate.

6.2 Requirements Traceability

As a basis for all validation and verification tasks all high-quality software must start with a definition of requirements. This means that each high-level software requirement must map to a lower level requirement, design, and implementation. The objective is to ensure that the complete system has been implemented as defined and that there is no surplus code. Terminology may vary regarding what different requirement tiers are called, but this fundamental element of sound software engineering practice remains.

Simply ensuring that system-level requirements map to something tangible in the requirements decomposition tree, design and implementation is not enough. The complete set of requirements comes from multiple sources, including system-level requirements, high-level requirements, and low-level (or derived) requirements. As illustrated below there is seldom a 1:1 mapping from system-level requirements to source code, so a traceability mechanism is required to map and record the dependency relationships of requirements throughout the requirements decomposition tree (Fig. 2).

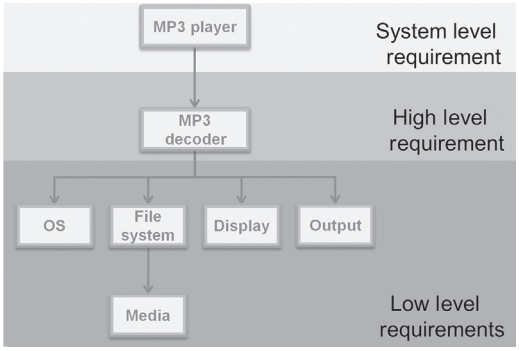


Fig. 2 Example of "1:Many" mapping from system-level requirements through a requirements decomposition tree.

To complicate matters further each level of requirements might be captured using a different mechanism. For instance, a formal requirements capture tool might be used for system-level requirements while high-level requirements are captured in PDF and low-level requirements captured in a spreadsheet.

Modern requirements traceability solutions enable mapping throughout these levels right down to the verification tasks associated with the source code. The screenshot (Fig. 3) shows an example of this. Using this type of requirements traceability tool the 100%

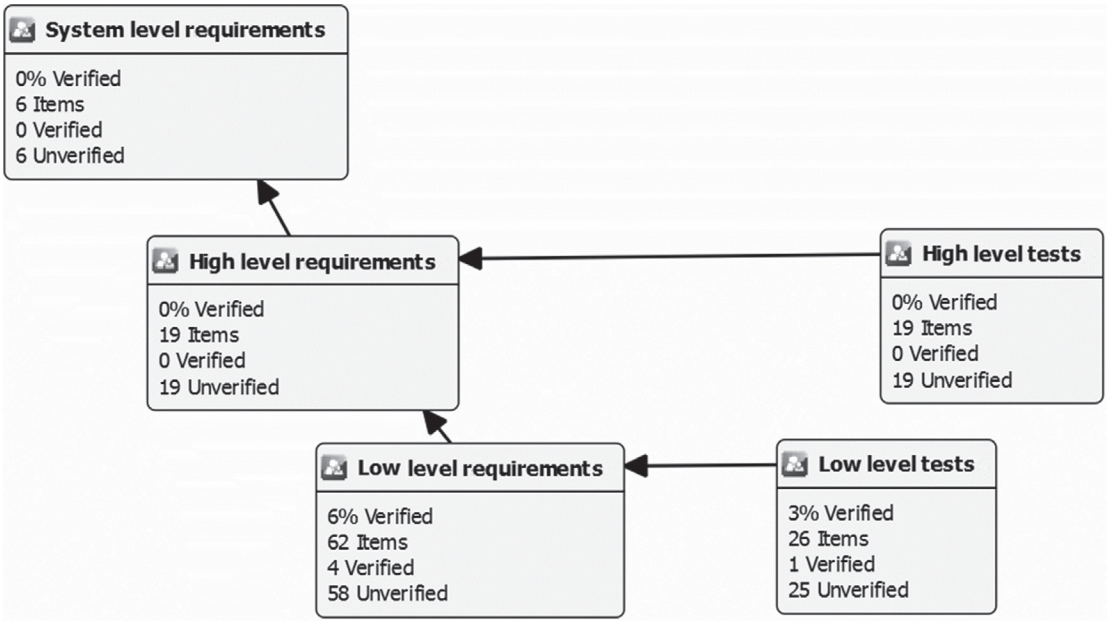


Fig. 3 Traceability from high-level requirements down to source code and verification tasks.

requirements coverage metric objective can clearly be measured, no matter how many layers of requirements, design, and implementation decomposition are used. This makes monitoring system completion progress an extremely straightforward activity.

It would be easy to overlook the requirements element of software development, but the fact is that even the best static and dynamic analysis in tandem do not prove that the software fulfills its requirements.

Widely accepted as a development best practice, bidirectional requirements traceability ensures that not only are all requirements implemented, but also that all development artifacts can be traced back to one or more requirements. Requirements traceability can also cover relationships with other entities such as intermediate and final work products, changes in design documentation, and test plans. Standards such as the automotive ISO 26262 or medical IEC 62304 demand bidirectional traceability, and place constant emphasis on the need for the complete and precise derivation of each development tier from the one above it.

When requirements are managed well, traceability can be established from the source requirement to its lower level requirements and from the lower level requirements back to their source. Such bidirectional traceability helps determine that all source requirements have been completely addressed and that all lower level requirements can be traced to a valid source.

Such an approach lends itself to a model of continuous and progressive use: first, of automated code review, followed by unit test, and subsequently system test with its execution tracing capability to ensure that all code functions exactly as the requirements dictate, even on the target hardware itself—a requirement for more stringent levels of most such standards.

While this is and always has been a laudable principle, last-minute changes to requirements or code made to correct problems identified during test tend to leave such ideals in disarray.

Despite good intentions many projects fall into a pattern of disjointed software development in which requirements, design, implementation, and testing artifacts are produced from isolated development phases. Such isolation results in tenuous links between requirements, the development stages, and/or the development teams.

The traditional view of software development shows each phase flowing into the next, perhaps with feedback to earlier phases, and a surrounding framework of configuration management and process (e.g., Agile, RUP). Traceability is assumed to be part of the relationships between phases. However, the reality is that, while each individual phase may be conducted efficiently, the links between development tiers become increasingly poorly maintained over the duration of projects.

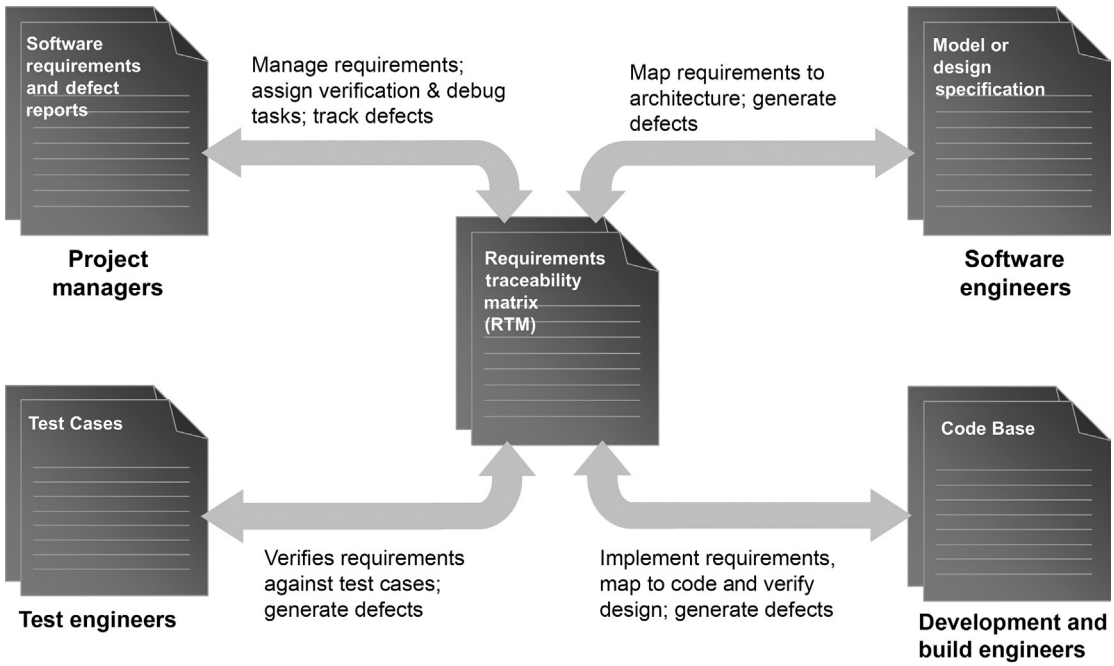


Fig. 4 The RTM sits at the heart of the project defining and describing the interaction between the design, code, test, and verification stages of development.

The answer to this conundrum lies in the requirements traceability matrix (RTM) that sits at the heart of any project even if it is not identified as such (see Fig. 4). Whether or not the links are physically recorded and managed they still exist. For example, a developer creates a link simply by reading a design specification and using that to drive the implementation.

Safety- and security-critical standards dictate that requirements should be traceable down to high-level code and in some cases object code, but elsewhere more pragmatism is usually required. A similar approach can be taken for any project with varying levels of detail depending on criticality both of the project as a whole and within an individual project. The important factor is to provide a level of traceability that is adequate for the circumstance.

This alternative view of the development landscape illustrates the importance that should be attached to the RTM. Due to this fundamental centrality it is vital that project managers place sufficient priority on investing in tooling for RTM construction. The RTM must also be represented explicitly in any life cycle model to emphasize its importance (as Fig. 5 illustrates). With this elevated focus the RTM is constructed and maintained efficiently and accurately.

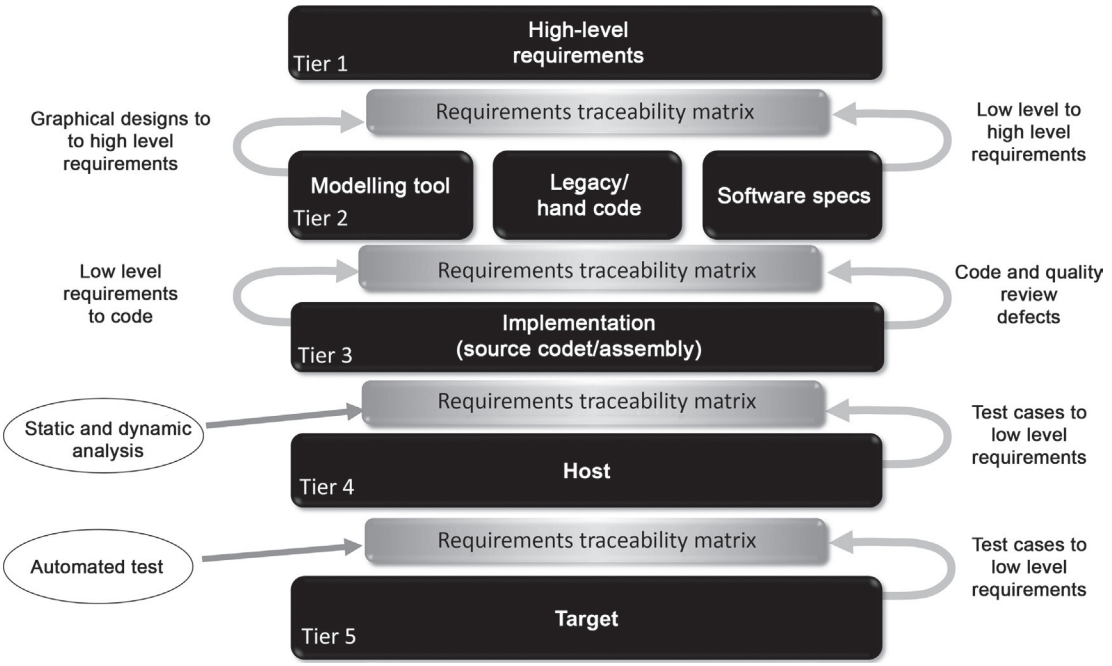


Fig. 5 The RTM plays a central role in a development life cycle model. Artifacts at all stages of development are linked directly to the requirements matrix, and changes within each phase automatically update the RTM so that overall development progress is evident from design through coding and test.

When the RTM becomes the center of the development process it impacts on all stages of design from high-level requirements through to target-based deployment. Where an application is safety critical each tier is likely to be implemented in full, but once again a pragmatic interpretation of the principles can be applied to any project.

Tier 1 high-level requirements might consist of a definitive statement of the system to be developed. This tier may be subdivided depending on the scale and complexity of the system.

Tier 2 describes the design of the system level defined by Tier 1. Above all, this level must establish links or traceability with Level 1 and begin the process of constructing the RTM. It involves the capture of low-level requirements that are specific to the design and implementation and have no impact on the functional criteria of the system.

Tier 3's implementation refers to the source/assembly code developed in accordance with Tier 2. Verification activities include code rule checking and quality analysis. Maintenance of the RTM presents many challenges at this level as tracing requirements to source code files may not be specific enough and developers may need to link to individual functions.

In many cases the system is likely to involve several functions. The traceability of those functions back to Tier 2 requirements includes many-to-few relationships. It is very easy to overlook one or more of these relationships in a manually managed matrix.

In **Tier 4** formal host-based verification begins. Once code has been proven to meet the relevant coding standards using automated code review, unit, then integration and system tests may be included in a test strategy that may be top-down, bottom-up, or a combination of both. Software simulation techniques help create automated test harnesses and test case generators as necessary, and execution histories provide evidence of the degree to which the code has been tested. Such testing could be supplemented with robustness testing if required, perhaps by means of the automatic definition of unit test vectors or by static prediction of dynamic behavior.

Test cases from Tier 4 should be repeatable at Tier 5, if required.

This is the stage that confirms the software functions as intended within its development environment, even though there is no guarantee it will work when in its target environment. Testing in the host environment first allows the time-consuming target test to merely confirm that the tests remain sound in the target environment.

Tier 5's target-based verification represents the on-target testing element of formal verification. This frequently consists of a simple confirmation that the host-based verification performed previously can be duplicated in the target environment, although some tests may only be applicable in that environment itself.

Where reliability is paramount and budgets permit, the static analysis of dynamic behavior with its "full range" data sets would undoubtedly provide a complementary tool for such an approach. However, dynamic analysis would remain key to the process.

6.3 Static Analysis—Adherence to a Coding Standard

One of the most basic attributes of code that affects quality is readability. The more readable a piece of code is, the more testable it is. The more testable it is, the more likely it will have been tested to a reasonable level of completion. Unfortunately, as The International Obfuscated C Code Contest has demonstrated, there are many ways to create complex and unreadable code for the simplest of applications. This metric is about adopting even a basic coding standard to help enhance code quality by establishing the rules for a minimum level of readability for all the code created within a project.

Modern coding standards go way beyond just addressing readability, however. Encapsulating the wisdom and experience of their creators, coding standards, such as the Motor Industry Software

Reliability Association (MISRA) C and C++ coding standards and the JSF Airborne Vehicle C++ standard or the Barr group (formerly Netrino) Embedded C Coding standard, also identify specific code constructs that can affect overall code quality and reliability, such as areas of C or C++ that the ISO standards state are either undefined or implementation specific.

Coding standards, such as the CERT-C or C++ Secure Coding Standards and the Common Weakness Enumeration list (CWE) also help to identify code constructs that can lead to potentially exploitable vulnerabilities in code.

The optimum coding standard for a project will depend on the project objectives. Fig. 6 provides a simple outline of the objectives for several coding standards.

In practice, most projects will create their own custom standard that uses one or more of these as a baseline, and modify the standard to suit their particular needs. Clearly, software that is safe AND secure is often desirable! Fortunately, the same attributes that make code safe very frequently also make it secure, and it is no coincidence that the MISRA organization, for example, have always focused their guidelines on critical systems, rather than safety-critical systems. The 2016 release of MISRA C:2012 AMD1, "Additional security guidelines for MISRA C:2012," further reinforced that position.

One area, in particular, where these reliability and security-oriented coding standards excel is in identifying code constructs that lead to latent defects, which are defects that are not normally detected during the normal software verification process yet reveal themselves once the product is released. Consider the following simple example:

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  #define MAX_SIZE 16U
5
6  int32_t main(void)
7  {
8    uint16_t theArray[MAX_SIZE];
9    uint16_t idx;
10   uint16_t *p_var;
```

Coding standard	Language	Objective/application
MISRA	C & C++	High reliability software
JSF Airborne Vehicle C++ Standard	C++	High reliability software
Barr Group Embedded C Standard	C	Defect free Embedded C Code
CERT Secure C Coding Standard	C	Secure Software

Fig. 6 Outline of objectives for several popular coding standards.

```
11   uint16_t UR_var;
12
13   p_var = &UR_var;
14
15   for(idx = 0U; idx < MAX_SIZE; idx += *p_var;)
16   {
17     theArray[idx] = 1U;
18   }
19
20   for(idx = 0U; idx <= MAX_SIZE; idx++)
21   {
22     printf(" %d", theArray[idx]);
23   }
24
25   return(0);
26 }
```

It compiles without warnings using either GCC or Microsoft Visual Studio (the latter requires the user to provide a stdint.h implementation for versions earlier than 2010). On inspection an experienced programmer can find the errors in this fairly simple code which contains both an array-out-of-bounds error (an off-by-one error on line 20 when variable idx == MAX_SIZE) and a reference to an uninitialized variable (on line 11, and line13 for the loop counter increment operator). These violate MISRA C:2012 rules 18.1 (relating to out-of-bounds errors) and 9.1 (which prohibits an object from being read before it has been set), respectively.

At its most basic an array-out-of-bounds error is a buffer overflow, even though it is legal C and/or C++ code. For secure code, buffer overflow is one of the most common vulnerabilities leading to the worst possible type of exploit: the execution of arbitrary code.

Nondeterminism is also a problem with the uninitialized variables in the example. It is impossible to predict the behavior of an algorithm when the value of its variables cannot be guaranteed. What makes this issue even worse is that some compilers will actually assign default values to uninitialized variables! For example, the Microsoft Visual Studio compiler assigns the value 0xCCCC to the variable UR_var by default in debug mode. While this value is meaningless in the context of the algorithm above, it is deterministic so the code will always behave in the same way. Switch to release mode, however, and the value will be undefined resulting in nondeterministic behavior.

Although real code is never as straightforward as this, even in this example there is some isolation from the latter issue as pointer aliasing is used to reference the UR_var variable.

A further MISRA C:2012 violation (rule 14.2: "A *for* loop shall be well-formed") relates to the complexity of the loop counter in this example. It is legitimate C code, but is difficult to understand and hence potentially error prone (Fig. 7).

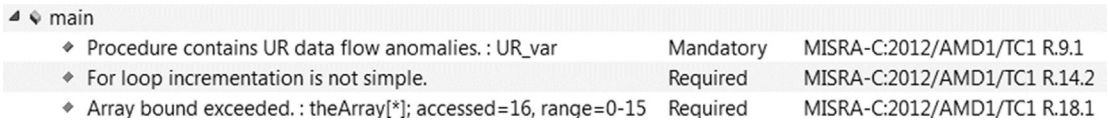


Fig. 7 Static analysis results showing enforcement of the MISRA C:2012 coding standard for the above code, revealing several violations.

In addition to the obvious benefits of identifying and eliminating latent defects the most significant additional benefit of using static analysis tools for coding standards enforcement is that it helps peer review productivity. By ensuring that a piece of code submitted for peer review has contravened no mandatory rules the peer review team can get away from focusing on minutiae and on to what they do best: ensuring that the implementations under inspection are fit for purpose and the best that they can be.

Another area where enforcement of coding standards is beneficial is in identifying unnecessary code complexity.

Complexity is not in and of itself a bad thing; complex problems require complex solutions, but code should not be more complex than necessary. This leads to sections of code that are unnecessarily difficult to read, even more difficult to test, and as a result have higher defect rates than a more straightforward equivalent implementation. It follows that unnecessary complexity is to be avoided.

Several coding standards incorporate maximum code complexity limits as a measure for improving overall code quality. The following case study explains cyclomatic complexity and knots metrics and how they may be used to show how complex a function is. The case study goes on to explain how essential cyclomatic complexity and essential knots metrics can show whether or not a function has been written in a structured manner, in turn giving a measure of complexity.

6.3.1 Essential Knots and Essential Cyclomatic Complexity—Case Study

6.3.1.1 Basic Blocks and Control Flow Branches

It is initially useful to consider how the construction of high-level software code can be described.

A “basic block” is a sequence of one or more consecutive, executable statements in a source program such that the sequence has a start point, an end point, and no internal branches.

In other words, once the first executable statement in a basic block is executed, then all subsequent statements in that basic block can be assumed to be executed in sequence.

Control flow branches and natural succession provide the links between basic blocks.

6.3.1.2 Control, Static, and Dynamic Flow Graphs

The logical flow of source, object, or assembler code can be represented by a control flow graph as first conceived by American computer scientist Frances Elizabeth "Fran" Allen. A control flow graph consists of a number of representations of basic blocks (represented by circular "nodes") interconnected with arrowed lines to represent the decision paths (called "links").

During the static analysis of source code it is possible to detect and understand the structure of the logic associated with it.

Control flow graphs form the basis of static flow graphs to show code structure. Dynamic flow graphs superimpose that representation of the code with execution history information to show which parts have been executed. In both cases the exact color-coding and symbolism used to represent different aspects of these graphs varies between tools—a fact further complicated in this book by the absence of color!

In the following static and dynamic flow graphs diamond-shaped nodes are used where a basic block includes a function call; otherwise circles are used. Relevant use of shading is explained in the context of each illustration.

In languages such as C and C++ it is often necessary to reformat the code to show only one instruction per line. This circumvents the problem of nomenclature for identifying decision points and instructions that occur within a single line of source code (Fig. 8).

6.3.1.3 Calculating a Knots Value

A knot is a point where two control flows intersect.

Knot analysis measures the amount of disjointedness in the code and hence the amount of "jumping about" a code reader will be required to undertake. An excessive number of knots may mean that a program can be reordered to improve readability and reduce complexity. A knot is not in itself a "bad thing," and knots appear in many perfectly acceptable constructs such as for, while, if/else, switch, and exception (Fig. 9).

Because they are a function of the chosen programming style and high-level language the number of knots in a function gives an indication of the complexity added to it as a result of program implementation. An excessive number of knots implies unnecessary complexity.

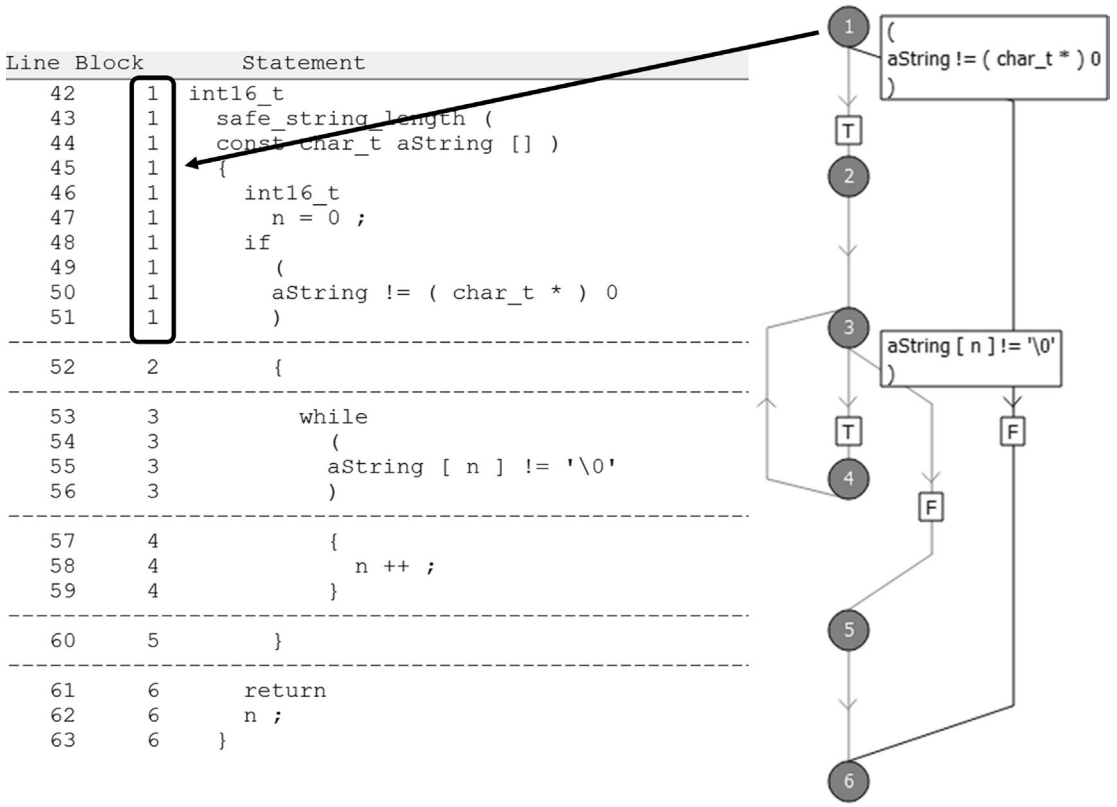


Fig. 8 Reformatted code shows the basic blocks ("nodes") connected by branches ("links").

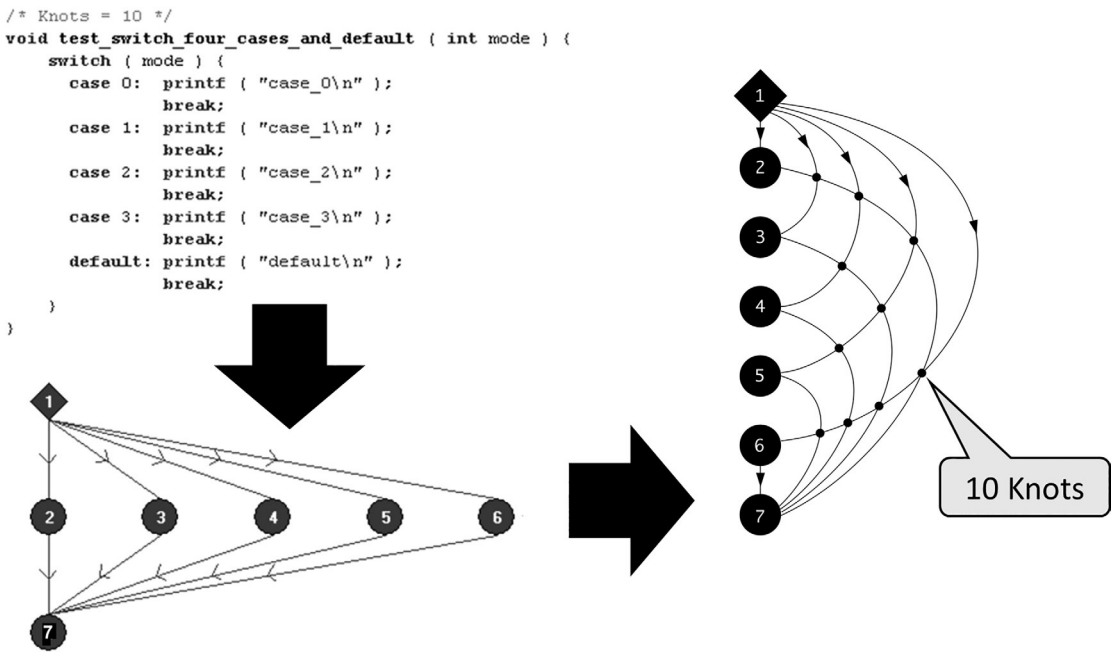


Fig. 9 A switch statement generally generates a high number of knots. There are 10 in this example, as shown on the right-hand side.

6.3.1.4 Calculating a Cyclomatic Complexity Value

Cyclomatic complexity is another measure of how complex a function is. It is a value derived from the geometry of the static flow graph for the function. The absolute value itself is therefore a little abstract and meaningless in isolation, but it provides a comparator to show the relative complexity of the problem addressed by one function vs. another.

Cyclomatic complexity is represented algebraically by the nomenclature $V(G)$ and can be derived in a number of ways, the simplest perhaps being a count of the number of "regions" separated by the links and nodes of the control or static flow graph (Fig. 10).

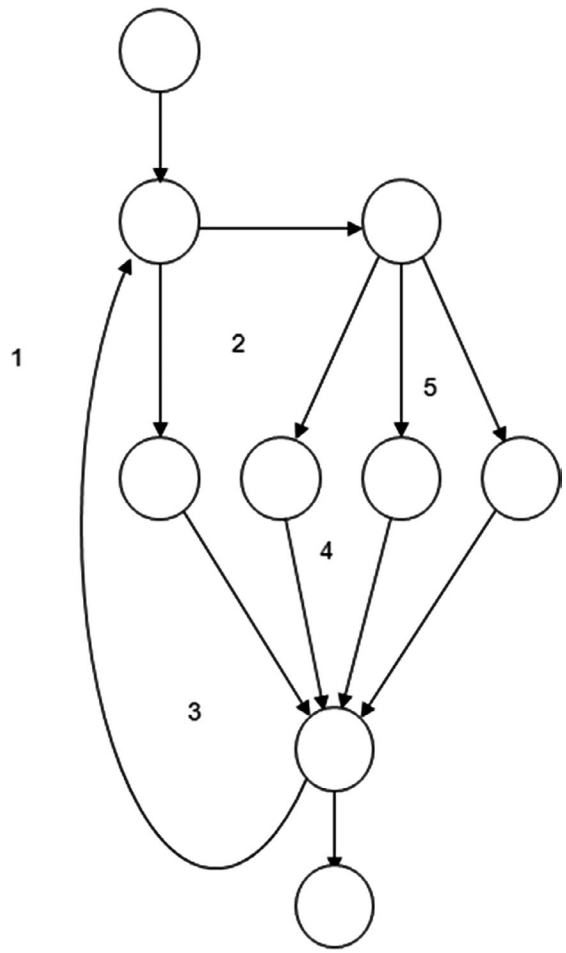


Fig. 10 An example of cyclomatic complexity derivation from a control flow graph. Here the cyclomatic complexity value $V(G) = 5$.

6.3.1.5 Identifying Structured Programming Templates—Structured Analysis

The concept of "structured" programming has been around since the 1960s, derived particularly from work by Böhm and Jacopini, and Edsger Dijkstra. In its modern implementation "structured elements" are defined as those constructs within the code that adhere to one of six "structured programming templates" as illustrated in Fig. 11.

Structured analysis is an iterative process where the static flow graph is repeatedly assessed to see whether it is possible to match one of the structured programming templates to a part of it. If so, that is "collapsed" to a single node as illustrated in Fig. 12.

The process is repeated until no more templates can be matched.

6.3.1.6 Essential Knots and Essential Cyclomatic Complexity

If this modified static flow graph is then used as the basis for knots and cyclomatic complexity calculations the resulting metrics are known as essential knots and essential cyclomatic complexity, respectively.

If there is only one node on any static flow graph it will exhibit no knots and only one region, meaning that a perfectly structured function will always have no essential knots and a cyclomatic complexity of 1.

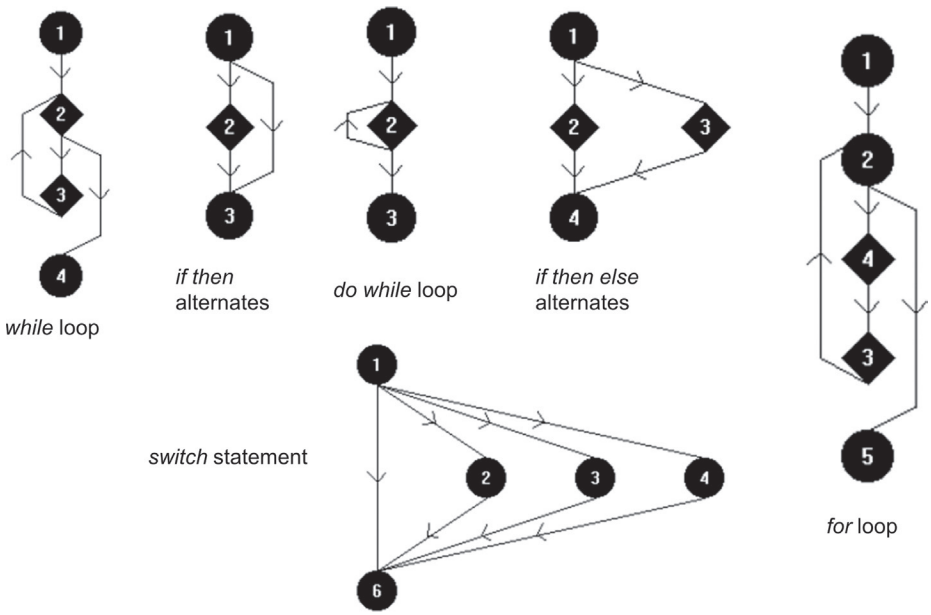


Fig. 11 The six structured programming templates.

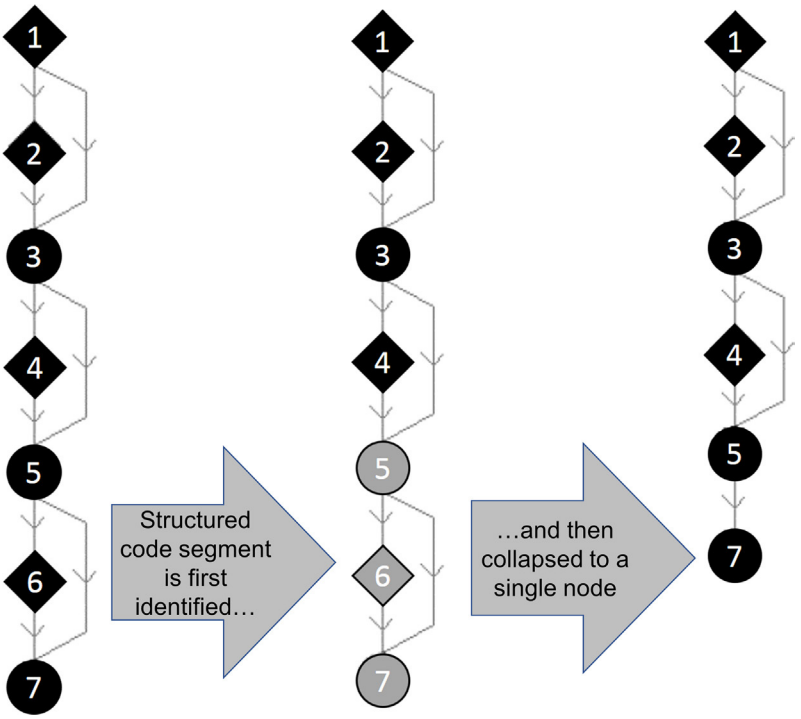


Fig. 12 Performing structured analysis.

The converse of this "perfect" result is that the essential measures will be greater than 0 and 1, respectively, showing that the code is not structured and hence may be unnecessarily complex.

6.4 Understanding Dynamic Analysis

As previously discussed, dynamic analysis involves the execution of some or all of the application source code. It is useful to consider some of the more widely used techniques that fall within this domain.

One such technique is a system-level functional test that defines perhaps the oldest test genre of them all. Simply described, when the code is written and completed the application is exercised using sample data and the tester confirms that everything works as it should.

The problem with applying this approach in isolation is that there is no way of knowing how much of the code has actually been exercised. Structural coverage analysis addresses this problem by reporting which areas of the application source code have been exercised by the test data and, more importantly, which areas have not. In its

simplest form structural coverage analysis is reported in the form of statement coverage. More sophisticated reporting mechanisms can then build upon this to report coverage of decision points and perhaps even control flow paths.

Unit test is another widely used dynamic analysis technique that has been around almost as long as software development itself. The cornerstone of this technique at its most basic is that each application building block (unit)—an individual procedure, function, or class—is built and executed in isolation from test data to make sure that it does just what it should do without any confusing input from the remainder of the application.

To support the necessary isolation of this process there needs to be a harness program to act as a holding mechanism that calls the unit, details any included files, "stubs" any procedure called by the unit, and prepares data structures for the unit under test to act upon.

Not only is creating that harness from first principles a laborious task, but it also takes a lot of skill. More often than not the harness program requires at least as much testing as the unit under test.

Perhaps more importantly, a fundamental requirement of software testing is to provide an objective, independent view of the software. The very intimate code knowledge required to manually construct a harness compromises the independence of the test process, undermining the legitimacy of the exercise.

6.5 The Legacy From High-Integrity Systems

In developing applications for the medical, railway, aerospace, and defense industries, unit test is a mandatory part of a software development cycle—a necessary evil. For these high-integrity systems, unit test is compulsory and the only question is how it might be completed in the most efficient manner possible. It is therefore no coincidence that many of the companies developing tools to provide such efficiency have grown from this niche market.

In more mundane environments, perceived wisdom is often that unit testing is a nice idea in principle, but commercially unjustifiable. A significant factor in that stance is the natural optimism that abounds at the beginning of any project. At that stage why would anyone spend money on careful unit testing? There are great engineers in the team, the design is solid, and sound management is in place. What could possibly go wrong?

However, things can and do go wrong, and while unit test cannot guarantee success it can certainly help to minimize failure. It therefore makes sense to consider the principles proven to provide quick and easy unit tests in high-integrity systems in the context of less demanding environments.

6.6 Defining Unit, Module, and Integration Test

For some the terms "unit test" and "module test" are synonymous. For others the term "unit" implies the testing of a single procedure, whereas "module" suggests a collection of related procedures, perhaps designed to perform some particular purpose within the application.

Using the latter definitions manually developed module tests are likely to be easier to construct than unit tests, especially if the module represents a functional aspect of the application itself. In this case most of the calls to procedures are related and the code accesses related data structures, which makes the preparation of the harness code more straightforward.

Test tools render the distinction between unit and module tests redundant. It is perfectly possible to test a single procedure in isolation and equally possible to use the exact same processes to test multiple procedures, a file or multiple files of procedures, a class (where appropriate), or a functional subset of an entire system. As a result the distinction between unit and module test is one that has become increasingly irrelevant to the extent that the term "unit test" has come to include both concepts.

This flexibility facilitates progressive integration testing. Procedures are first unit-tested and then collated as part of the subsystems, which in turn are brought together to perform system tests.

It also provides options when a pragmatic approach is required for less critical applications. A single set of test cases can exercise a specified procedure, all procedures called as a result of exercising the single procedure as illustrated in Fig. 13, or anything in between. The use of test cases that prove the functionality of the whole call chain are easily constructed. Again, it is easy to "mix and match" the processes depending on the criticality of the code under review.

6.7 Defining Structural Coverage Analysis

The structural coverage analysis approach is all about ensuring that enough testing is performed on a system to meet its quality objectives.

For the most complete testing possible it is necessary to ensure that every possible execution path through the code under test is executed at least once. In practice, this is an unachievable aim. An observation made by G.J. Myers in 1976 explains why this is so; Myers described a 100-line program that had 10^{18} unique paths. For comparative purposes he noted that the universe is only about 4×10^{17} s old. With this observation Myers concluded that complete software execution path testing is impossible, so an approximation alternative and another metric are required to assess testing completeness.

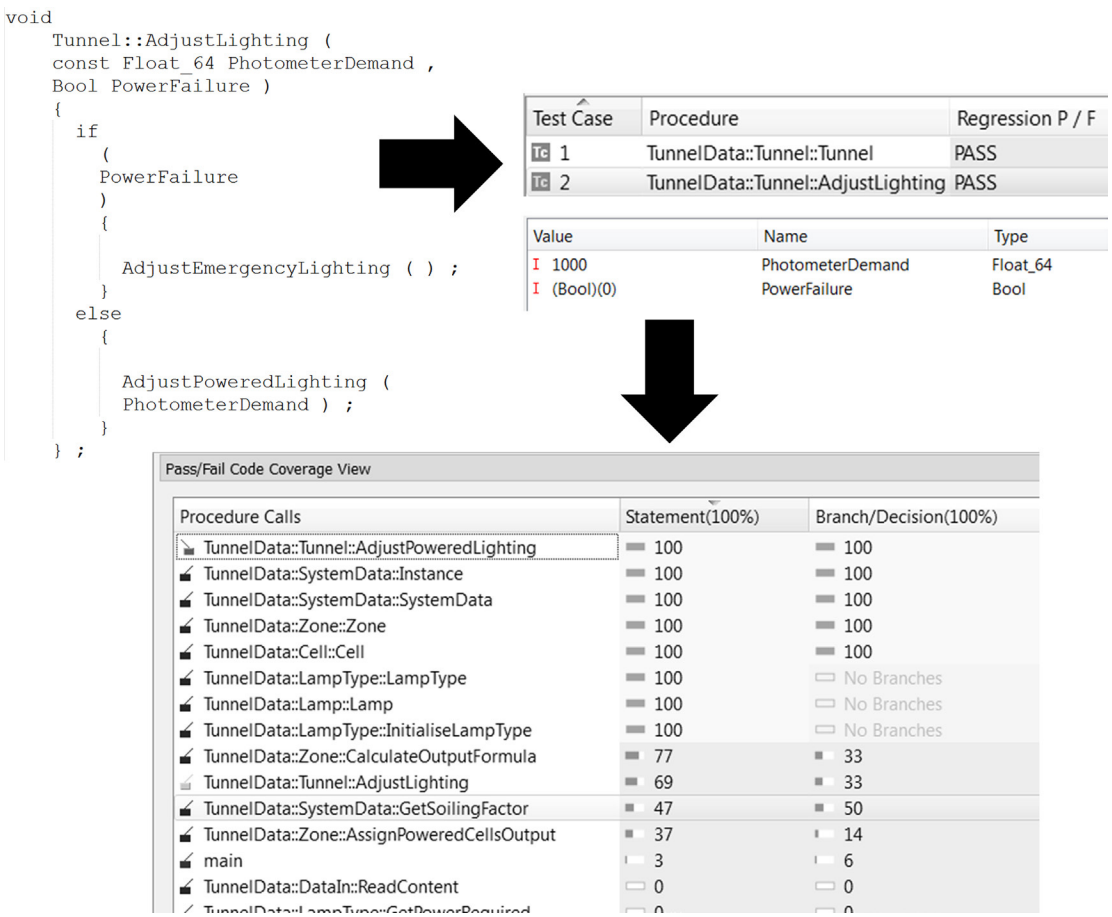


Fig. 13 A single test case (inset) can exercise some or all of the call chain associated with it. In this example "AdjustLighting" is the subject of the test.

Structural coverage analysis has proven to be an excellent technique for that purpose.

The closest structural coverage analysis metric to the 100% execution path ideal is based on the linear code sequence and jump (LCSAJ) software analysis technique, or jump-to-jump path (JJ-path) coverage as it is sometimes described. LCSAJ analysis identifies sections of code that have a single input path and a single output path, referred to as an interval. Within each interval each possible execution path is then identified. A structural coverage analysis metric is then determined by measuring which of these possible execution paths within an interval have been executed.

As with all these metrics the use of tools for measuring structural coverage analysis greatly increases measurement efficiency,

```
void
    Tunnel::AdjustLighting (
    const Float_64 PhotometerDemand ,
    Bool PowerFailure )
    {
      if
        (
        PowerFailure
        )
        {

          AdjustEmergencyLighting ( ) ;
        }
      else
        {

          AdjustPoweredLighting (
          PhotometerDemand ) ;
        }
    } ;
```

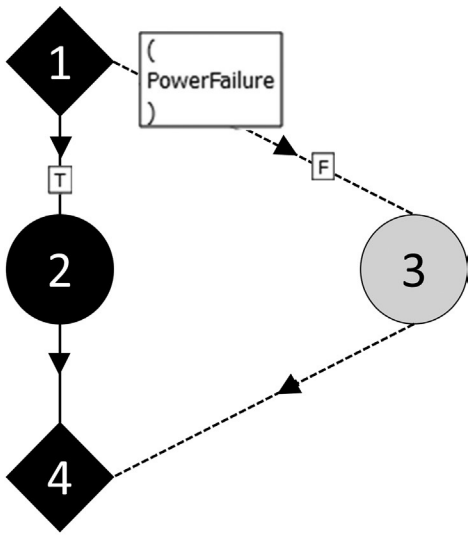


Fig. 14 Example coverage analysis results presented against a control flow graph. Black nodes and solid lines represent exercised code to simulate the color-coding used in test tools.

effectiveness, and accuracy. In addition, the visualization of results provides excellent feedback on what additional test cases are required to improve overall coverage measurements. Test tools generally use coloring to represent coverage information. In Fig. 14 and elsewhere in this chapter black nodes and solid branch lines represent exercised code.

From these results it is a straightforward exercise to determine which test data need to be generated to exercise the remaining "cold" paths, making the ability to generate the quality-oriented reports required for certification extremely straightforward.

6.8 Achieving Code Coverage With Unit Test and System Test in Tandem

Traditionally, many applications have been tested by functional means only—and no matter how carefully the test data are chosen the percentage of code actually exercised can be very limited.

That issue is compounded by the fact that the procedures tested in this way are only likely to handle data within the range of the current application and test environment. If anything changes a little—perhaps in the way the application is used or perhaps as a result of slight modifications to the code—the application could be running an entirely untested execution path in the field.

Of course, if all parts of the system are unit-tested and collated on a piecemeal basis through integration testing, then this will not happen. But what if timescales and resources do not permit such an exercise?

The more sophisticated unit test tools provide the facility to instrument code. This instrumented code is equipped to "track" execution paths, providing evidence of the parts of the application that have been exercised during execution. Such an approach provides the information to produce data such as those depicted in Fig. 14.

Code coverage is an important part of the testing process in that it shows the percentage of the code that has been exercised and proven during test. Proof that all code has been exercised correctly need not be based on unit tests alone. To that end some unit tests can be used in combination with system tests to provide a required level of execution coverage for a system as a whole.

Unit tests can complement system tests to execute code that would not normally be exercised in the running of the application. Examples include defensive code (e.g., to prevent crashes due to inadvertent division by zero), exception handlers, and interrupt handlers.

6.8.1 Unit Test and System Test in Tandem—Case Study

Consider the following function, taken from a lighting system written in C++. Line 7 includes defensive code designed to ensure that a divide by zero error cannot occur:

```
1  Sint_32 LampType::GetPowerRequired(const Float_64 LumensRequired) const
2  /* Assume a linear deterioration of efficiency from HighestPercentOutput lm/W output from each lamp at
3  maximum output, down to LowestPercentOutput lm/W at 20% output. Calculate power required based on
4  the resulting interpolation. */
5 {
6    Sint_32 Power=0;
7     if (((mMaximumLumens-mMinimumLumens)>Small) && LumensRequired>=mMinimumLumens))
8        {
9          Power = (Sint_32)(mMinimumPower + (mMaximumPower-mMinimumPower)*
10                ((LumensRequired-mMinimumLumens)/(mMaximumLumens-mMinimumLumens)));
11       }
12       return Power;
13 }
```

The dynamic flow graph for this function after system test shows that most of the statements and control flow decisions have been exercised as part of system test. However, in a correctly configured system the values of "mMaximumLumens" and "mMinimumLumens" will never be similar enough to force the defensive aspect of the code to be exercised (Fig. 15).

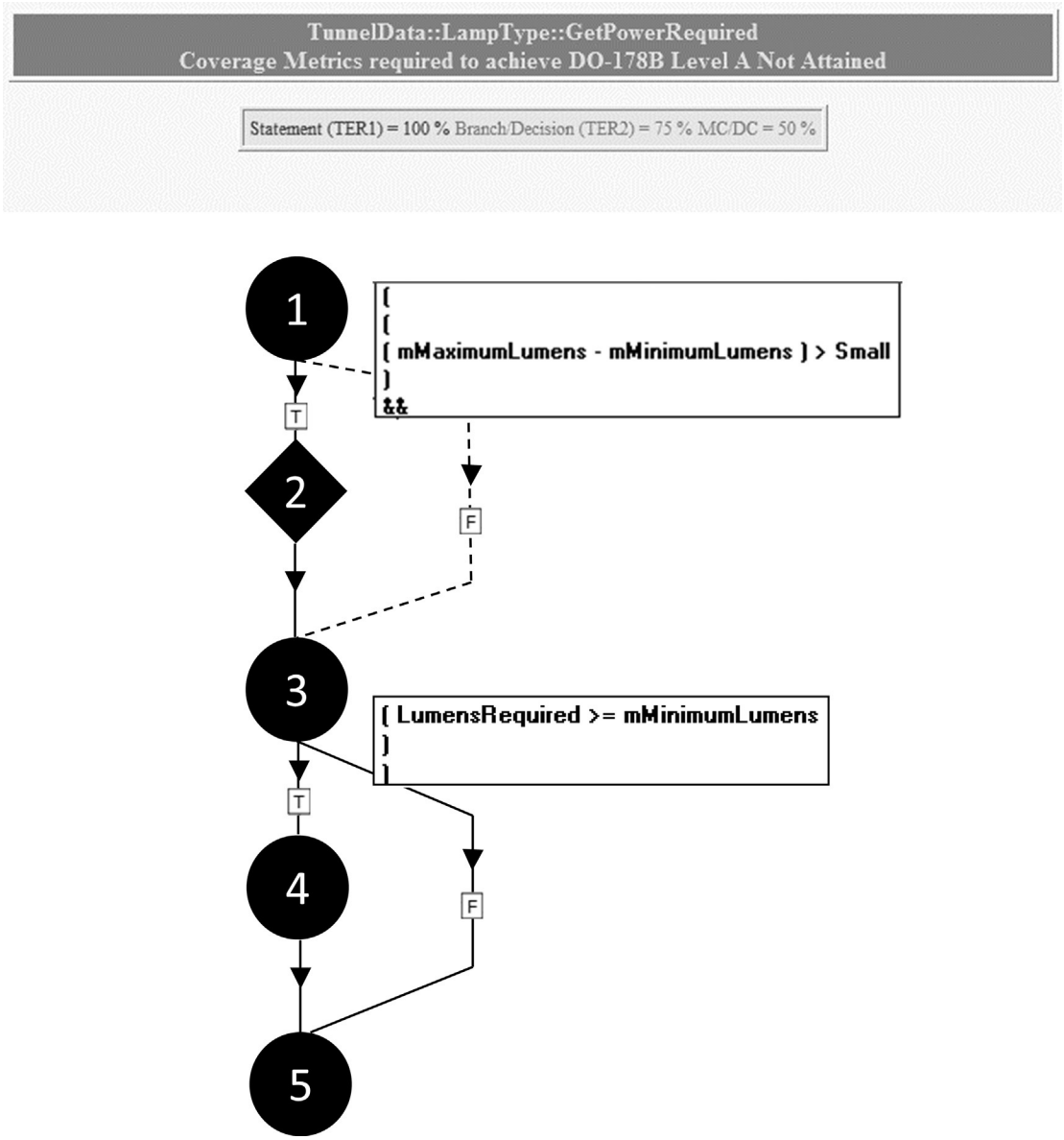


Fig. 15 After system test most of the function has been exercised, but a branch associated with defensive programming remains.

Unit test can be used to complement the code coverage achieved during system test, which forces the defensive branch to be taken (Fig. 16).

The coverage from the unit test and system test can then be combined so that full coverage is demonstrated (Fig. 17).

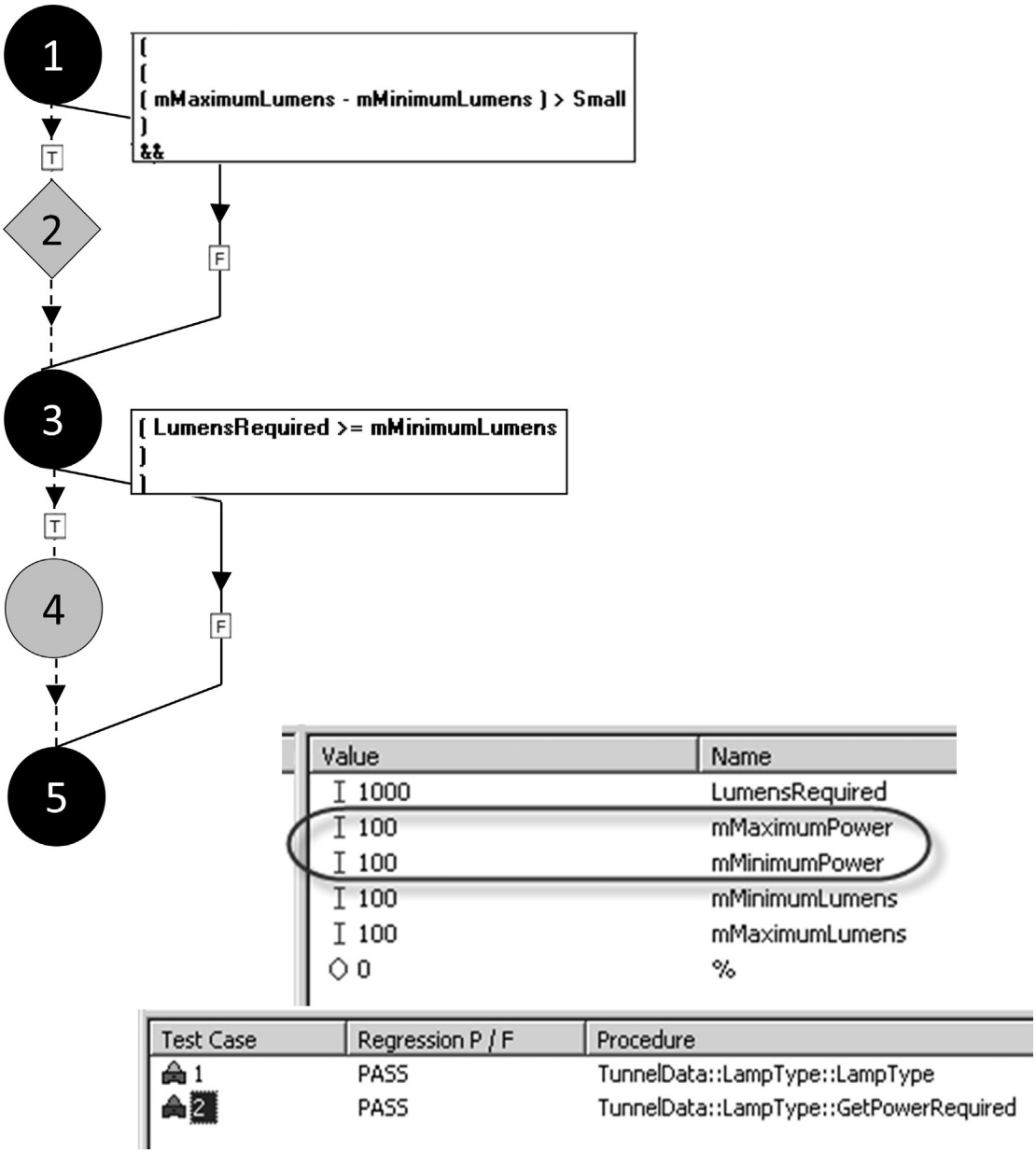


Fig. 16 Unit test exercises the defensive branch left untouched by system test.

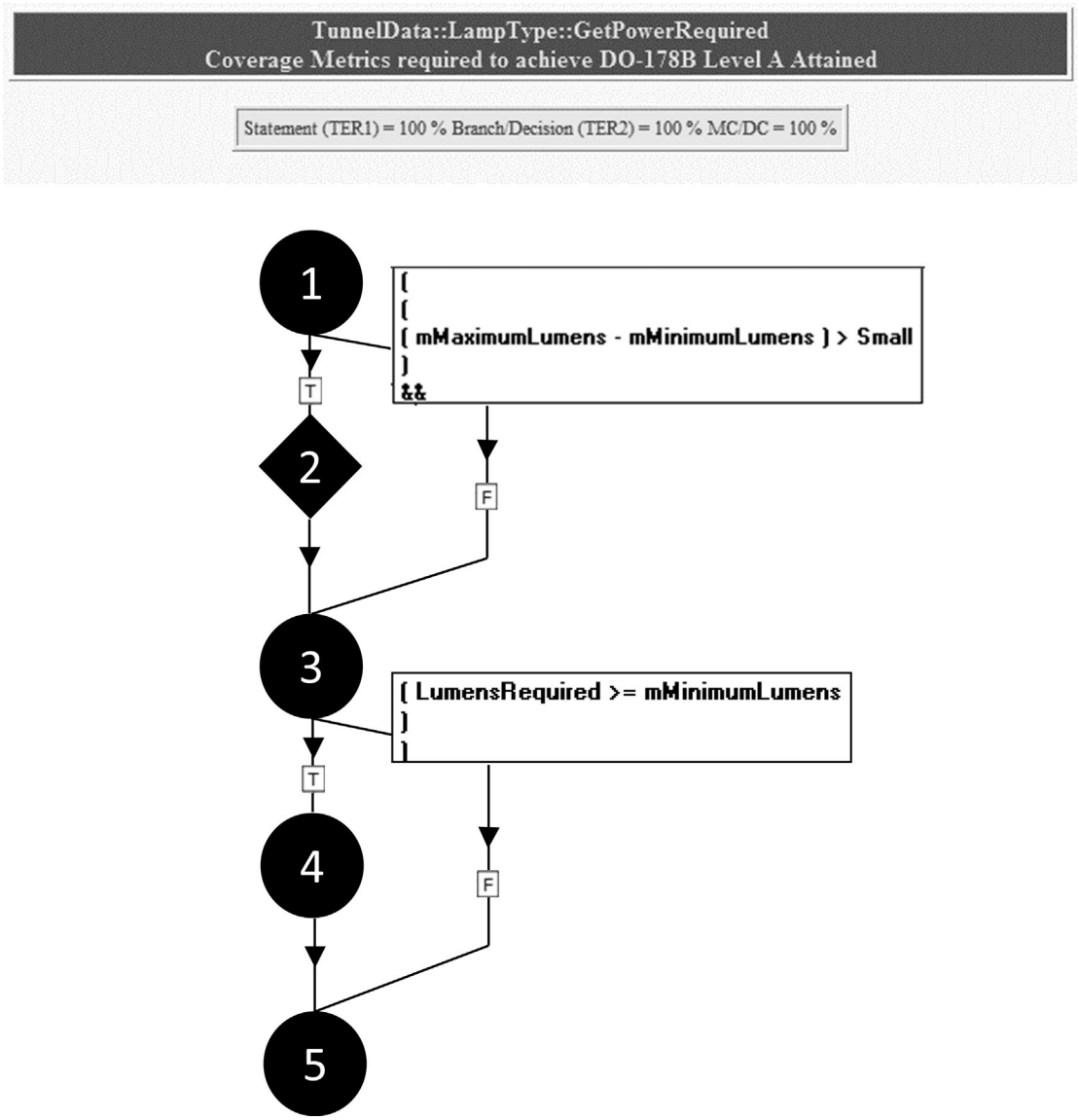


Fig. 17 Full coverage is demonstrated by combining the system test and unit test.

6.9 Using Regression Testing to Ensure Unchanged Functionality

During the course of development, ongoing development can compromise the functionality of software that is considered complete.

As software evolves it is therefore essential to keep reapplying existing tests and monitor the subsequent test outcomes against previously determined expected results. This is a process known as regression testing. Often this is achieved by using test case files to store sequences of tests, and it is then possible to recall and reapply them to any revised code to prove that none of the original functionality has been compromised.

Once configured, more sophisticated regression test processes can be initiated as a background task and run perhaps every evening. Reports can highlight any changes to the output generated by earlier test runs. In this way any code modifications leading to unintentional changes in application behavior can be identified and rectified immediately and the impact of regression tests against other, concurrent, test processes can be kept to a minimum.

Modern unit test tools come equipped with user-friendly, point-and-click graphical user interfaces, which are easy and intuitive to use. However, a GUI interface is not always the most efficient way to implement the thousands of test cases likely to be required in a full-scale development. In recognition of this, the more sophisticated test tools are designed to allow these test case files to be directly developed from applications such as Microsoft Excel. As before, the "regression test" mechanism can then be used to run the test cases held in these files.

6.10 Unit Test and Test-Driven Development

In addition to using unit test tools to prove developed code they can also be used to develop test cases for code that is still in the conception phase—an approach known as test-driven development (TDD). As illustrated, TDD is a software development technique that uses short development iterations based on prewritten unit test cases that define desired improvements or new functions. Each iteration produces code necessary to pass the set of tests that are specific to it. The programmer or team refactors the code to accommodate changes (Fig. 18).

6.11 Automatically Generating Test Cases

Unit tests are usually performed to demonstrate adherence to requirements, to show that elements of the code perform the function they were designed to perform.

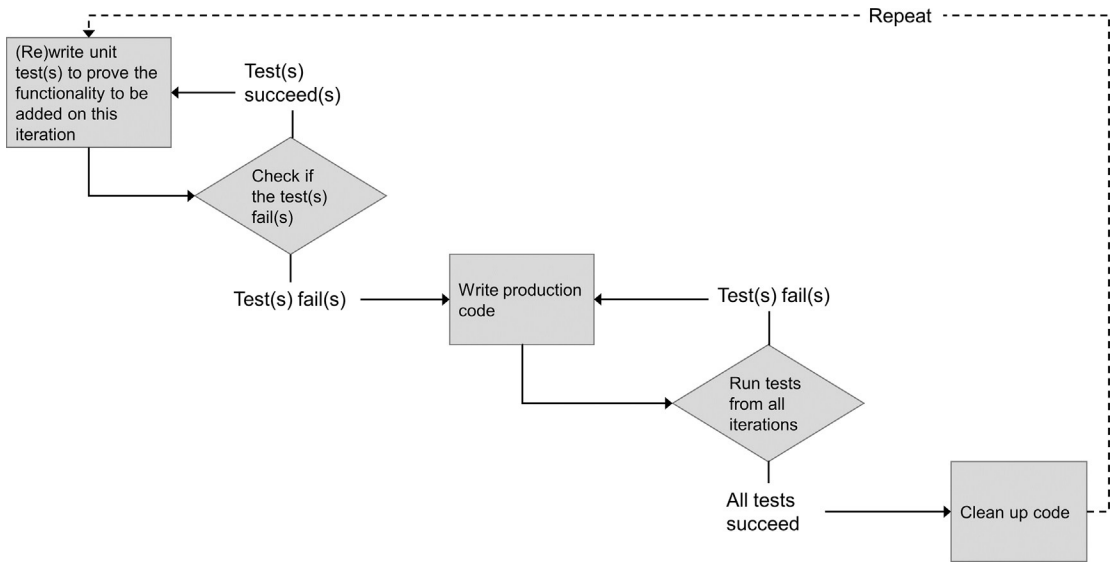


Fig. 18 Unit test tools lend themselves admirably to test-driven development by providing a mechanism to write test cases before any source code is available.

Generally, then, the output data generated through unit tests are important in themselves, but this is not necessarily always the case.

There may be occasions when the fact that the unit tests have successfully been completed is more important than the test data themselves. To address these circumstances as efficiently as possible, the more sophisticated unit test tools can generate test cases automatically, based on information gleaned by means of the initial static analysis of the software under test. For example:

- Source code may be required to pass robustness tests.
- The functionality of source code may already be proven, but the required level of code coverage is unsatisfied.
- A "personality profile" of source code may be required prior to the modification of source code. Sequences of test cases can be generated based on the unchanged code and then exercised again when the source has been modified to prove that there has been no inadvertent detrimental effect on existing functionality.

To tune test cases generated in this way, tools provide a range of options to allow different aspects of the code to be considered. For example, options may include

- generation of test cases to exercise upper and lower boundary values;
- generation of minimum/mean/maximum values; and
- generation of the optimal number of test cases in maximizing code coverage.

6.11.1 A Word of Caution

It would be easy to view the automatic generation of test cases as a potential "silver bullet," an answer to all possible test questions with the minimum of effort.

It certainly represents an easy way to generate test cases, although it does require caution to ensure, for instance, that test cases do not result in infinite loops or null pointers.

However, there is an intrinsic compromise in the basic premise. The tests themselves are based on the source code—not on any external requirements. Detractors of the technique might argue that automatically generated test cases prove only that the source code does what it was written to do and that they would prove nothing if interpretation of the requirements were to be fundamentally flawed.

It is clearly true that such a test process compromises the principle of test independence and it is certainly not being suggested that automatically generated tests can or should replace functional testing, either at the system or unit test level.

However, once the test cases have been generated they can be executed in an identical manner to that provided for conventionally defined unit tests. The input to and output from each of the test cases are available for inspection, so that the correctness of the response from the software to each generated case can be confirmed if required.

7 Setting the Standard

Recent quality concerns are driving many industries to start looking seriously at ways to improve the quality of software development. Not surprisingly, there are marked differences not only in the quality of software in the different sectors, but also in the rate of change of that quality. For example, both the railway and process industries have long had standards governing the entire development cycle of electrical, electronic, and programmable electronic systems, including the need to track all requirements. On the other hand, although a similar standard in the automotive sector was a relatively recent introduction, there has been an explosion in demand for very high–quality embedded software as it moves toward autonomous vehicle production.

The connectivity demanded by autonomous vehicles has also become a significant factor across other safety-critical sectors, too. The advent of the Industrial Internet of Things (IIoT), connected medical devices, and connectivity in aircraft has seen a new emphasis on a need for security in parallel with functional safety.

Clearly, a safety-critical device cannot be safe if it is vulnerable to attack, but the implications go beyond that. For example, the

vulnerability of credit card details from an automotive head unit do not represent a safety issue, but demand a similar level of care in software development to ensure that they are protected.

In short, the connected world poses threats to product safety and performance, data integrity and access, privacy, and interoperability.

7.1 The Terminology of Standards

In layman's terms there are documents that define how a process should be managed and standards that dictate the instructions and style to be used by programmers in the process of writing the code itself.

These groups can be further subdivided. For example, there are many collections of these instructions for the use of development teams looking to seek approval for their efforts efficiently. But what are these collections of rules called collectively?

Unfortunately, there is little consensus for this terminology among the learned committees responsible for what these documents are actually called.

The MISRA C:2012 document, for example, is entitled "Guidelines for the use of the C language in critical systems" and hence each individual instruction within the document is a "guideline." These guidelines are further subdivided into "rules" and "directives."

Conversely, the HICC++ document is known as a "coding standards manual" and calls each individual instruction a "rule."

To discuss and compare these documents it is therefore necessary to settle on some umbrella terminology to use for them collectively. For that reason this chapter refers to "Process standards," "Coding standards," and "Coding rules" throughout and distinguishes "internal standards" used within an organization from "recognized standards" such as those established by expert committees.

7.2 The Evolution of a Recognized Process Standard

It is interesting to consider the evolution of the medical software standard IEC 62304 because it mirrors earlier experience in many other sectors.

The US government is well aware of the incongruence of the situation and is considering ways to counter it with the Drug and Device Accountability Act (http://www.govtrack.us/congress/bill.xpd?bill=s111-882). Recently, the FDA took punitive action against Baxter Healthcare and their infusion pumps, which the FDA has forced the company to recall (https://www.lawyersandsettlements.com/lawsuit/baxter-colleague-infusion-pumps-recall.html).

The net result is that many medical device providers are being driven to improve their software development processes as a result of commercial pressures. In short, they are doing so because it affects the "bottom line."

A common concept in the standards applied in the safety-critical sectors is the use of a tiered, risk-based approach for determining the criticality of each function within the system under development (Fig. 19). Typically known as safety integrity levels there are usually four or five grades used to specify the necessary safety measures to avoid an unreasonable residual risk of the whole system or a system component. The SIL is assigned based on the risk of a hazardous event occurring depending on the frequency of the situation, the impact of possible damage, and the extent to which the situation can be controlled or managed (Fig. 20).

For a company to make the transition to developing certified software they must integrate the standard's technical safety requirements into their design. To ensure that a design follows the standard a company must be able to outline the fulfillment of these safety requirements from design through coding, testing, and verification.

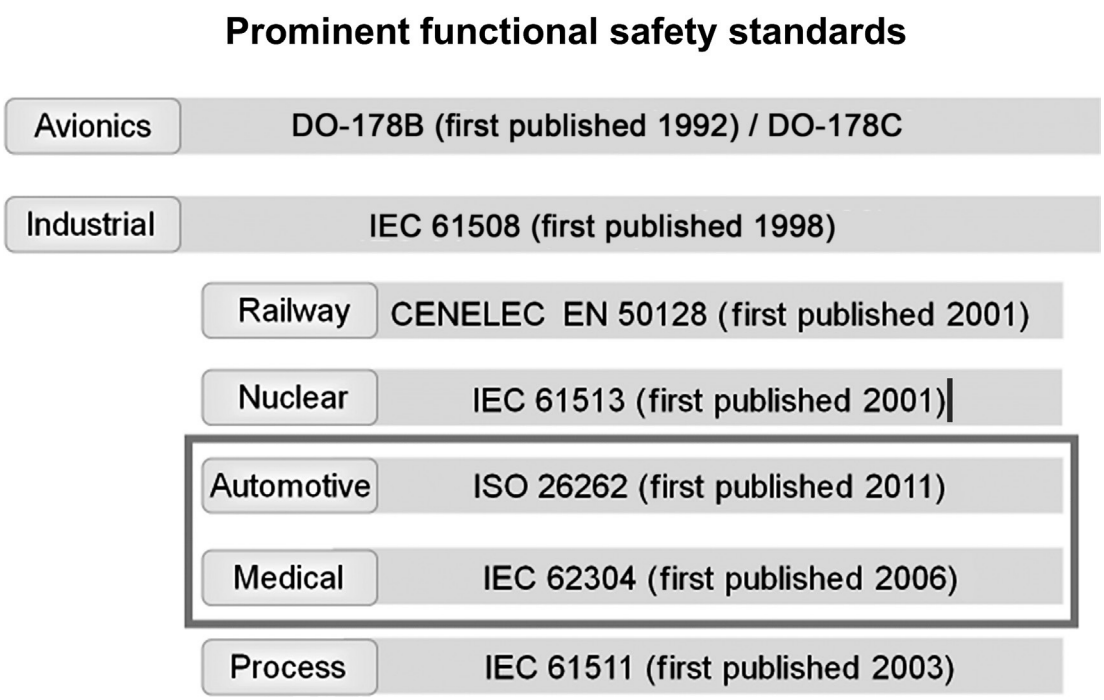


Fig. 19 Many safety-critical standards have evolved from the generic standard IEC 61508. Sectors that are relative newcomers to this field (highlighted) were advantaged by the availability of established and proven tools to help them achieve their goals.

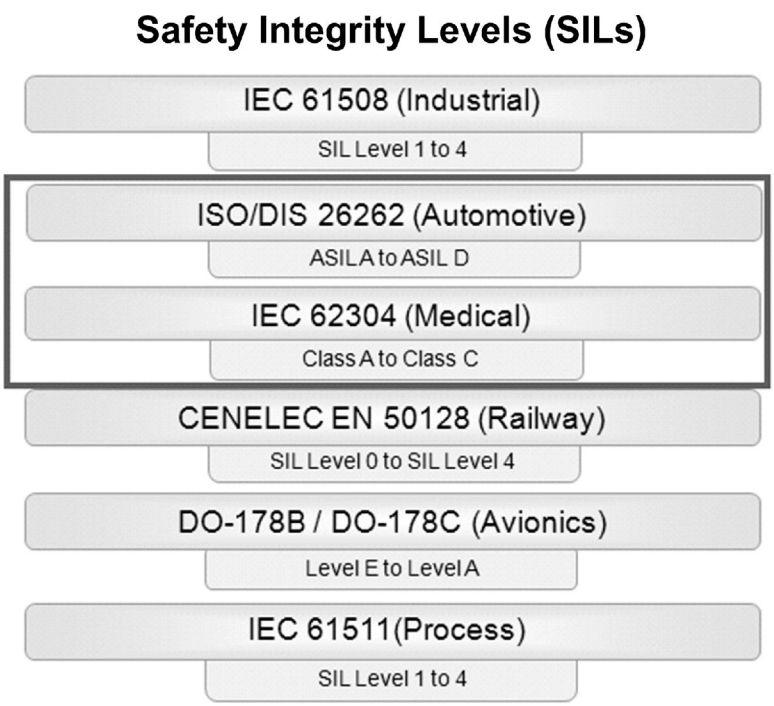


Fig. 20 The example standards all apply some concept of safety integrity levels, although the terminology used to describe them varies. Again, the principles hold true in the highlighted sectors where standards are newest.

To ease the adoption of this standard and manage the shift in requirements many companies use gap analysis. Gap analysis begins by gathering and analyzing data to gauge the difference between where the business is currently and where it wants to be. Gap analysis examines operating processes and artifacts generated, typically employing a third party for the assessment. The outcome will be notes and findings on which the company or individual project may act.

7.2.1 ISO 26262 Recognized Process Standard—Case Study

In parallel with the advances in the medical device sector and in response to the increased application of electronic systems to automotive safety-critical functions the ISO 26262 standard was created to comply with needs specific to the application sector of electrical/electronic/programmable electronic (E/E/PE) systems within road vehicles.

In addition to its roots in the IEC 61508 generic standard it has much in common with the DO-178B/DO-178C standards seen in aerospace applications. In particular, the requirement for MC/DC (Modified Condition/Decision Coverage—a technique to dictate the

tests required to adequately test lines of code with multiple conditions) and the structural coverage analysis process is very similar.

Safety is already a significant factor in the development of automobile systems. With the ever-increasing use of E/E/PE systems in areas such as driver assistance, braking and steering systems, and safety systems this significance is set to increase.

The standard provides detailed industry-specific guidelines for the production of all software for automotive systems and equipment, whether it is safety critical or not. It provides a risk management approach including the determination of risk classes (automotive safety integrity levels, ASILs).

There are four levels of ASILs (A–D in ISO 26262) to specify the necessary safety measures for avoiding an unreasonable residual risk, with D representing the most stringent level.

The ASIL is a property of a given safety function—not a property of the whole system or a system component. It follows that each safety function in a safety-related system needs to have an appropriate ASIL assigned, with the risk of each hazardous event being evaluated based on the following attributes:

- frequency of the situation (or "exposure")
- impact of possible damage (or "severity")
- controllability.

Depending on the values of these three attributes the appropriate ASIL for a given functional defect is evaluated. This determines the overall ASIL for a given safety function.

ISO 26262 translates these safety levels into safety-specific objectives that must be satisfied during the development process. An assigned ASIL therefore determines the level of effort required to show compliance with the standard. This means that the effort and expense of producing a system critical to the continued safe operation of an automobile (e.g., a steer-by-wire system) is necessarily higher than that required to produce a system with only a minor impact in the case of a failure (e.g., the in-car entertainment system).

The standard demands a mature development environment that focuses on requirements that are specified in this standard. To claim compliance to ISO 26262 most requirements need to be formally verified, aside from exceptional cases where the requirement does not apply or where noncompliance is acceptable.

Part 4 of the standard concerns product development at the system level, and Part 6 of the standard concerns product development at the software level. The scope of these documents may be mapped on to any process diagram such as the familiar "V" model (Fig. 21).

Software analysis, requirements management, and requirements traceability tools are usually considered essential for large, international, cost-critical projects.

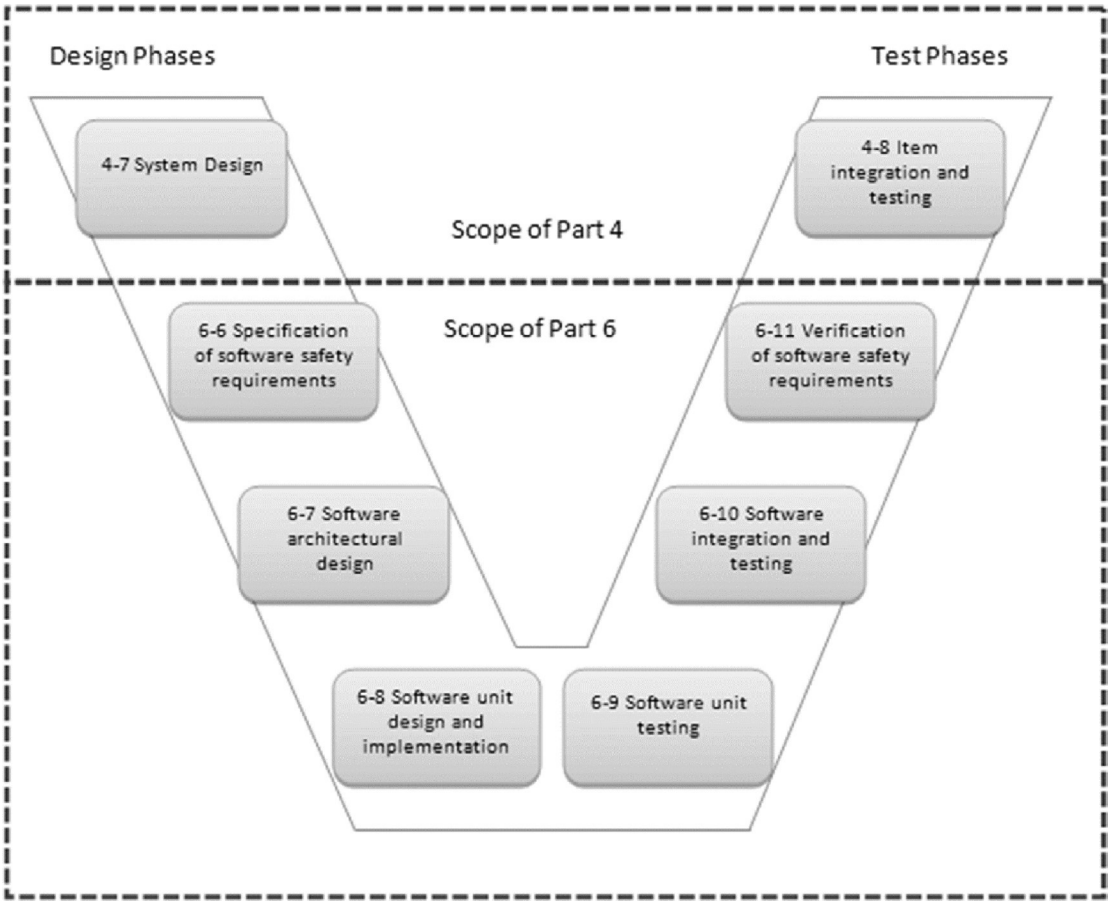


Fig. 21 Mapping the scope of ISO 26262 Part 4 and Part 6 onto the familiar "V" model.

7.2.2 ISO 26262 Process Objectives

ISO 26262 recognizes that software safety and security must be addressed in a systematic way throughout the software development life cycle. This includes the safety requirements traceability, software design, coding, and verification processes used to ensure correctness, control, and confidence both in the software and in the E/E/PE systems to which that software contributes.

A key element of ISO 26262 (Part 4) is the practice of allocating technical safety requirements in the system design and developing that design further to derive an item integration and testing plan and subsequently the tests themselves. It implicitly includes software elements of the system, with the explicit subdivision of hardware and software development practices being dealt with further down the "V" model.

ISO 26262 (Part 6) refers more specifically to the development of the software aspect of the product. It is concerned with:

- initiation of product development at the software level;
- derivation of software safety requirements from the system level (following from Part 4) and their subsequent verification;
- software architectural design;
- software unit design and implementation;
- software unit testing; and
- software integration and testing.

Traceability (or requirements traceability) refers to the ability to link system requirements to software safety requirements, and then software safety requirements to design requirements, and then to source code and associated test cases. Although traceability is not explicitly identified as a requirement in the main body of the text, it is certainly desirable in ensuring the verifiability deemed necessary in Section 7.4.2. Moreover, the need for "bidirectional traceability" (or upstream/downstream traceability) is noted in the same section.

7.2.3 Verification Tasks

The methods to be deployed in the development of an ISO 26262 system vary depending on the specified ASIL. This can be illustrated by reference to the verification tasks recommendations (as presented in Fig. 22).

Table 1 in Section 5.4.7 of Part 6 recommends that design and coding guidelines are used, and as an example cites the use of the MISRA C coding standard. It lists a number of topics that are to be covered by modeling and design guidelines. For instance, the enforcement of low complexity is highly recommended for all ASILs.

Modern test tools not only have the potential to cover all the obligatory elements for each ASIL, but they also have the flexibility in configuration to allow less critical code in the same project to be associated with less demanding standards. That principle extends to mixed C and C++ code, where appropriate standards are assigned to each file in accordance with its extension (Fig. 23).

Table 12 from Section 9 of Part 6 shows, for instance, that measuring statement coverage is highly recommended for all ASILs, and that branch coverage is recommended for ASIL A and highly recommended for other ASILs. For the highest ASIL D, MC/DC is also highly recommended.

Each of these coverage metrics implies different levels of test intensity. For example, 100% statement coverage is achieved when every statement in the code is executed at least once; 100% branch (or decision) coverage is achieved only when every statement is executed at least once, AND each branch (or output) of each decision is tested—that is, both false and true branches are executed.

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity	++ ✓	++✓	++✓	++✓
1b	Use of language subsets	++✓	++✓	++✓	++✓
1c	Enforcement of strong typing	++✓	++✓	++✓	++✓
1d	Use of defensive implementation techniques	o	+✓	++✓	++✓
1e	Use of established design principles	+✓	+✓	+✓	++✓
1f	Use of unambiguous graphical representation	+✓	++✓	++✓	++✓
1g	Use of style guides	+✓	++✓	++✓	++✓
1h	Use of naming conventions	++✓	++✓	++✓	++✓

”++” The method is highly recommended for this ASIL.

“+“ The method is recommended for this ASIL.

“o“ The method has no recommendation for or against its usage for this ASIL.

✓ Potential for efficiency gains through the use of test tools

Fig. 22 Mapping the potential for efficiency gains through the use of test tools to “ISO 26262 Part 6 Table 1: Topics to be covered by modeling and coding guidelines.”

Statement, branch, and MC/DC coverage can all be automated through the use of test tools. Some packages can also operate in tandem so that, for instance, coverage can be generated for most of the source code through a dynamic system test, and can be complemented using unit tests to exercise defensive code and other aspects that are inaccessible during normal system operation.

Similarly, Table 15 in Section 10.4.6 shows the structural coverage metrics at the software architectural level (Fig. 24).

Topics		ASIL			
		A	B	C	D
1a	Statement coverage	++✓	++✓	+✓	+✓
1b	Branch coverage	+✓	++✓	++✓	++✓
1c	MC/DC (modified condition/decision Coverage)	+✓	+✓	+✓	++✓

”++” The method is highly recommended for this ASIL.

“+“ The method is recommended for this ASIL.

“o“ The method has no recommendation for or against its usage for this ASIL.

✓ Potential for efficiency gains through the use of test tools

Fig. 23 Mapping the potential for efficiency gains through the use of test tools to “ISO 26262 Part 6 Table 14: Structural coverage metrics at the software unit level.”

Topics		ASIL			
		A	B	C	D
1a	Function coverage	+✓	+✓	++✓	++✓
1b	Call coverage	+✓	+✓	++✓	++✓

“++” The method is highly recommended for this ASIL.

“+” The method is recommended for this ASIL.

“o” The method has no recommendation for or against its usage for this ASIL.

✓ Potential for efficiency gains through the use of test tools

Fig. 24 Mapping the potential for efficiency gains through the use of test tools to the “ISO 26262 Part 6 Table 17: Structural coverage metrics at the software architectural level.”

7.2.4 SAE J3061 and ISO 26262

ISO 26262 requires any threats to functional safety to be adequately addressed, implicitly including those relating to security threats, but it gives no explicit guidance relating to cybersecurity.

At the time of ISO 26262's publication, that omission was perhaps to be expected. Automotive-embedded applications have traditionally been isolated, static, fixed function, device-specific implementations, and practices and processes have relied on that status. But the rate of change in the industry has been such that by the time of its publication in 2016, SAE International's Surface Vehicle Recommended Practice SAE J3061 was much anticipated.

Note the wording here. SAE J3061 is not a standard. It constitutes "recommended practice," and at the time of writing there are plans for it to be replaced by an ISO/SAE 21434 standard—the first result of a partnership between the ISO and SAE standards organizations.

SAE J3061 can be considered complementary to ISO 26262 in that it provides guidance on best development practices from a cybersecurity perspective, just as ISO 26262 provides guidance on practices to address functional safety.

It calls for a similar sound development process to that of ISO 26262. For example, hazard analyses are performed to assess risks associated with safety, whereas threat analyses identify risks associated with security. The considerations for the resulting security requirements for a system can be incorporated in the process described by ISO 26262 Part 8 Section 6: "Specification and management of safety requirements."

Another parallel can be found in the use of static analysis, which is used in safety-critical system development to identify constructs, errors, and faults that could directly affect primary functionality. In cybersecurity-critical system development, static code analysis is used instead to identify potential vulnerabilities in the code.

7.2.4.1 Beyond Functional Safety

Despite the clear synergy between the two standards it is important to note that SAE J3061 does more than simply formalize the need to include security considerations in functional safety requirements. It is easy to focus on an appropriate process once functional, safety, and security requirements are established, but the significance of malicious intent in the definition of those requirements should not be underestimated.

SAE J3061 emphasizes this point throughout. It argues that cybersecurity is likely to be even more challenging than functional safety, stating that "Since potential threats involve intentional, malicious, and planned actions, they are more difficult to address than potential hazards. Addressing potential threats fully, requires the analysts to think

like the attackers, but it can be difficult to anticipate the exact moves an attacker may make."

Perhaps less obviously, the introduction of cybersecurity into an ISO 26262-like formal development implies the use of similarly rigorous techniques into applications that are NOT safety critical—and perhaps into organizations with no previous obligation to apply them. SAE J3061 discusses privacy in general and personally identifiable information (PII) in particular, and highlights both as key targets for a bad actor of no less significance than the potential compromise of safety systems.

In practical terms, there is a call for ISO 26262-like rigor in the defense of a plethora of personal details potentially accessed via a connected car, including personal contact details, credit card and other financial information, and browse histories. It could be argued that this is an extreme example of the general case cited by the SAE J3061 standard, which states that "…there is no direct correspondence between an ASIL rating and the potential risk associated with a safety-related threat."

Not only does SAE J3061 bring formal development to less safety-critical domains, it also extends the scope of that development far beyond the traditional project development life cycle. Consideration of incident response processes, over-the-air (OTA) updates, and changes in vehicle ownership are all examples of that.

7.3 Freedom to Choose Adequate Standards

Not every development organization of every application is obliged to follow a set of process or coding standards that has been laid down by a client or a regulatory body. Indeed, it is probably reasonable to suggest that most are not in that position.

However, everyone wants their software to be as sound and as robust as possible. Even if it is a trivial application to be used by the writer as a one-off utility, no developer wants their application to crash. Even if such a utility expands into an application with wider uses, no one wants to have to deal with product recalls. No one wants to deal with irate end users. Even removing all external factors entirely, most people want the satisfaction of a job well done in an efficient manner.

So, it follows that if the use of process and coding standards is appropriate when safety or security issues dictate that software must be robust and reliable, then it is sensible to adopt appropriate standards even if an application is not going to threaten anyone's well-being if it fails.

Once that is established a sound pragmatic approach is required to decide what form those standards should take.

7.4 Establishing an Internal Process Standard

Many recognized standards are most ideally deployed within large organizations. There are many software development teams that

consist of two or three people all resident in the same office. It would clearly be overkill to deploy the same tools and techniques here as in a high-integrity team of hundreds of developers spread across the world.

That said, the principles of the requirements traceability matrix (RTM) established earlier in the chapter remain just as valid in either case. The difference lies in the scaling of the mechanisms and techniques used to confirm the traceability of requirements. For that reason an appropriate recognized process standard can prove very useful as a guideline for pragmatic application of similar principles in a less demanding environment.

7.4.1 Establishing a Common Foundation for an Internal Coding Rule Set

The principle of using a recognized standard as the basis for an internal one extends to the realm of coding standards.

Even where there is no legacy code to worry about there is frequently a good deal of inertia within a development team. Something as simple as agreement over the placement of brackets can become a source of great debate among people who prefer one convention over another.

Under these circumstances the establishment of a common foundation rule set that everyone can agree on is a sound place to begin.

For example, practically no one would advocate the use of the "goto" statement in C or C++ source code. It is likely that outlawing the use of the "goto" statement by means of a coding rule will achieve support from all interested parties. Consequently, it is an uncontentious rule to include as part of a common foundation rule set.

Establishing such a set of rules from nothing is not easy, and given that learned organizations are meeting regularly to do just that, neither is it a sensible use of resources.

It therefore makes sense to derive an internal standard from a recognized standard to which the organization might aspire in an ideal world, This does not necessarily imply an intention on the part of the development organization to ever fully comply with that standard, but it does suggest that the rules deployed as part of that subset will be coherent, complementary, and chosen with the relevant industry sector in mind.

7.4.2 Dealing With an Existing Code Base

This principle becomes a little more challenging when there is a legacy code base to deal with.

It is likely that the retrospective enforcement of a recognized coding standard, such as MISRA C:2012, to legacy code is too onerous and so a subset compromise is preferred. In that case it is possible to apply

a user-defined set of rules that could simply be less demanding or that could, say, place particular focus on portability issues.

Where legacy code is subject to continuous development 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 enable the correction of code to adhere to such rules as efficiently as possible. Some tools use a "drill down" approach to provide a link between the description of a violation in a report and an editor opened on the relevant line of code.

7.4.3 Deriving an Internal Coding Standard for Custom Software Development—Case Study

In many fields of endeavor for embedded software development—automobiles, aeroplanes, telephones, medical devices, weapons—the software life cycle contributes to a product life cycle of design and development, readying for production, then mass production. In some fields, such as control systems for bespoke plant or machinery, duplicates are the exception—not the rule. That brings a unique set of difficulties.

Imagine a situation where there is a software team of three or four developers within a larger engineering company. The actual value of the software within the context of a typical contract might be very small, but even so the software itself is critical to making sure that the client is satisfied when work is completed.

The original code has been designed to be configurable, but as each sale is made by the sales team the functionality is expanded to encompass new features to clinch the deal. The sales team is motivated by the commission derived from the whole of the contract, meaning that a modification to software functionality is not a primary concern for them.

The developers of the software team are tasked with implementing this expanding functionality for each new contract, often under great pressure from other areas of production.

Commercial milestones loom large for these hapless developers because, despite their small contribution to the overall value of the project, any failure on their part to meet the milestones could result in delay of a stage payment or the triggering of a penalty clause.

To make matters worse, numerous software developers have joined and departed from the team over the years. Each has had their own style and preferences and none has had the time to thoroughly document what they have done.

As a practical example, let us consider that the developers of such a code base, designed to control a widget production machine, have been tasked with establishing the quality of their software and improving it on an ongoing basis.

7.4.3.1 Establishing a Common Foundation

The first step in establishing an appropriate rule set is to choose a relevant recognized standard as a reference point—perhaps MISRA C++:2008 in this case.

Using a test tool the code base can be analyzed to discover the parts of the standard where the code base is already adequate. By opting to include information relating to all rules irrespective of whether the code base complies with them, a subset of the standard to which the code adheres can immediately be derived (Fig. 25).

Typically, the configuration facilities in the test tool can then be used to map the rules that have not been transgressed into a new subset of the reference standard, and the violated rules disabled as illustrated in Fig. 26.

7.4.3.2 Building Upon a Common Foundation

Even if nothing else is achieved beyond checking each future code release against this rule set, it is assured that the standard of the code in terms of adherence to the chosen reference standard will not deteriorate further.

However, if the aim is to improve the standard of the code, then an appropriate strategy is to review the violated rules. It is likely that there are far fewer different rules violated than there are individual violations, and test tools can sometime generate a useful breakdown of this summary information (Fig. 27).

In some cases it may be that a decision is reached that some rules are not appropriate and their exclusion justified.

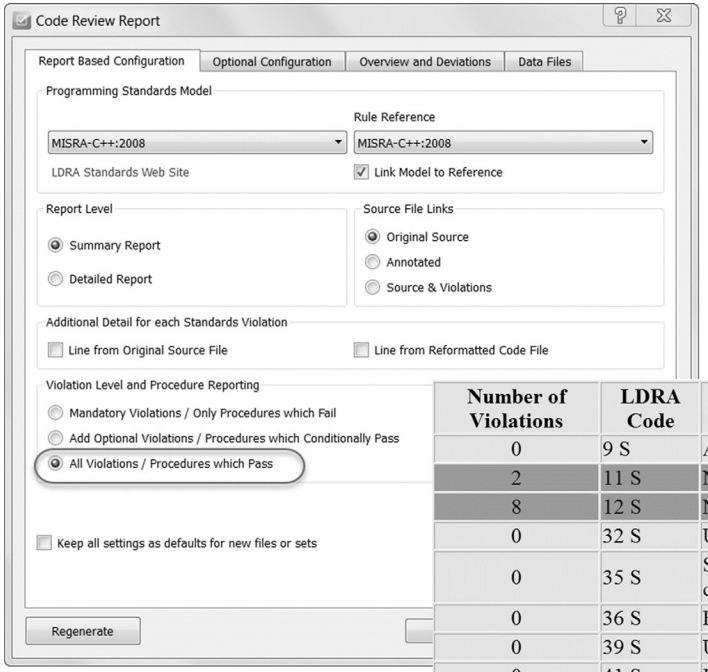
Prioritizing the introduction of the remaining rules can vary depending on the primary motivation. In the example it is obviously likely to be quicker to address the violations that occur once rather than those that occur 40 or 50 times, which will improve apparent adherence to the standard. However, it makes sense to initially focus on any particular violations that, if they were to be corrected, would address known functional issues in the code base.

Whatever the criteria for prioritization, 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.

8 Dealing With the Unusual

8.1 Working With Autogenerated Code

Many software design tools, such as IBM's Rhapsody and MathWork's Matlab, have the ability to automatically generate high-level source code from a UML or similar design model (Fig. 28).



Number of Violations	LDRA Code	Required Standards	MISRA-C++:2008 Code
0	9 S	Assignment operation in expression.	MISRA-C++:2008 5-0-1,6-2-1
2	11 S	No brackets to loop body.	MISRA-C++:2008 6-3-1
8	12 S	No brackets to then/else.	MISRA-C++:2008 6-4-1
0	32 S	Use of continue statement.	MISRA-C++:2008 6-6-3
0	35 S	Static procedure is not explicitly called in code analysed.	MISRA-C++:2008 0-1-1
0	36 S	Function has no return statement.	MISRA-C++:2008 8-4-3
0	39 S	Unsuitable type for loop variable.	MISRA-C++:2008 6-5-1
0	41 S	Ellipsis used in procedure parameter list.	MISRA-C++:2008 8-4-1
0	43 S	Use of setjmp/longjmp.	MISRA-C++:2008 17-0-5
7	44 S	Use of banned function, type or variable.	MISRA-C++:2008 17-0-1,18-0-2,18-2-1,18-4-1,19-3-1
0	47 S	Array bound exceeded.	MISRA-C++:2008 5-0-16
1	48 S	No default case in switch statement.	MISRA-C++:2008 6-4-6
12	49 S	Logical conjunctions need brackets.	MISRA-C++:2008 5-0-2,5-2-1
0	50 S	Use of shift operator on signed type.	MISRA-C++:2008 5-0-21
0	51 S	Shifting value too far.	MISRA-C++:2008 5-8-1
0	52 S	Unsigned expression negated.	MISRA-C++:2008 5-3-2

Fig. 25 Using source code violation reporting to identify rules that are adhered to. In this report subset only the rules highlighted have been violated.

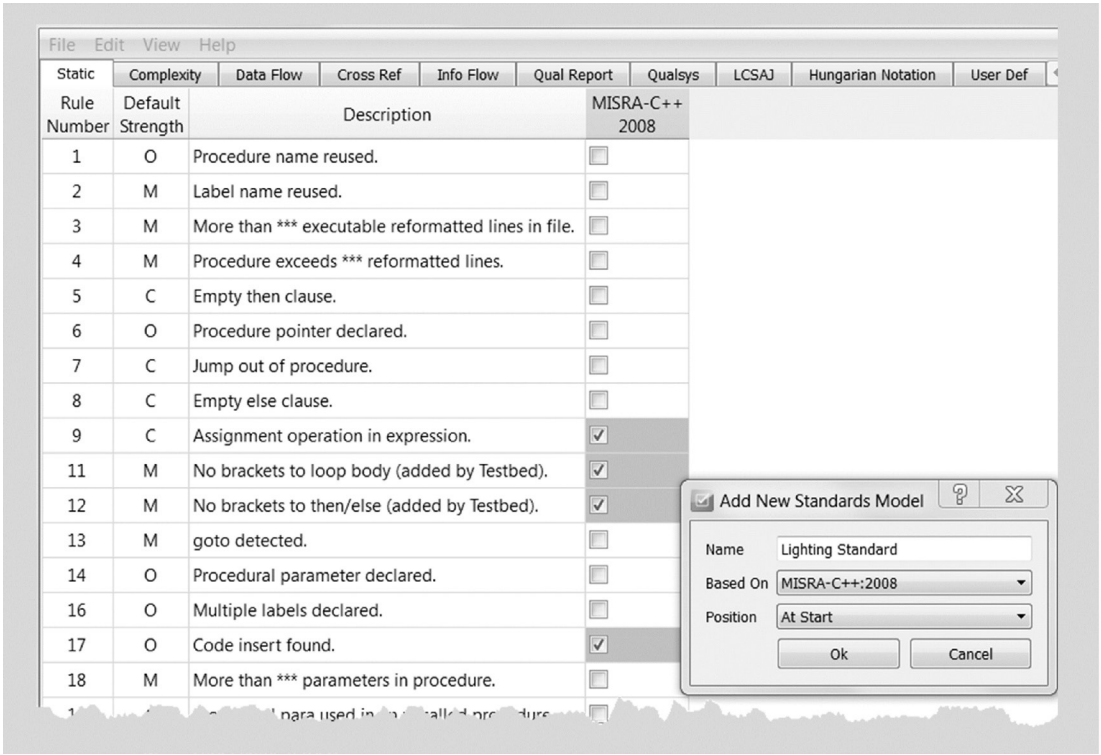


Fig. 26 Using unviolated rules from a recognized standard as the basis for an internal standard.

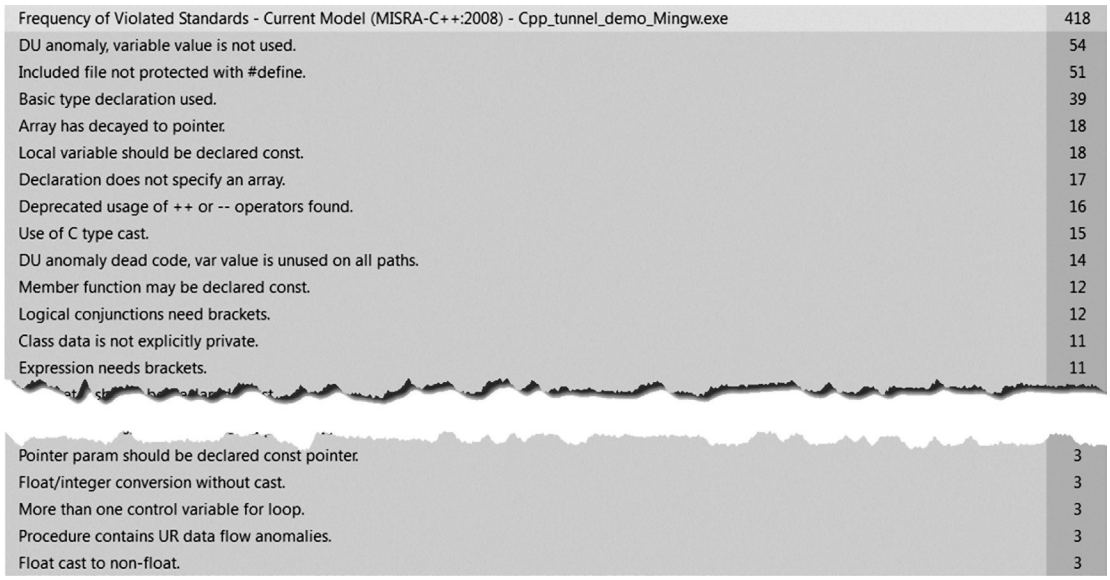


Fig. 27 Summary showing the breakdown of rule violations in a sample code set.

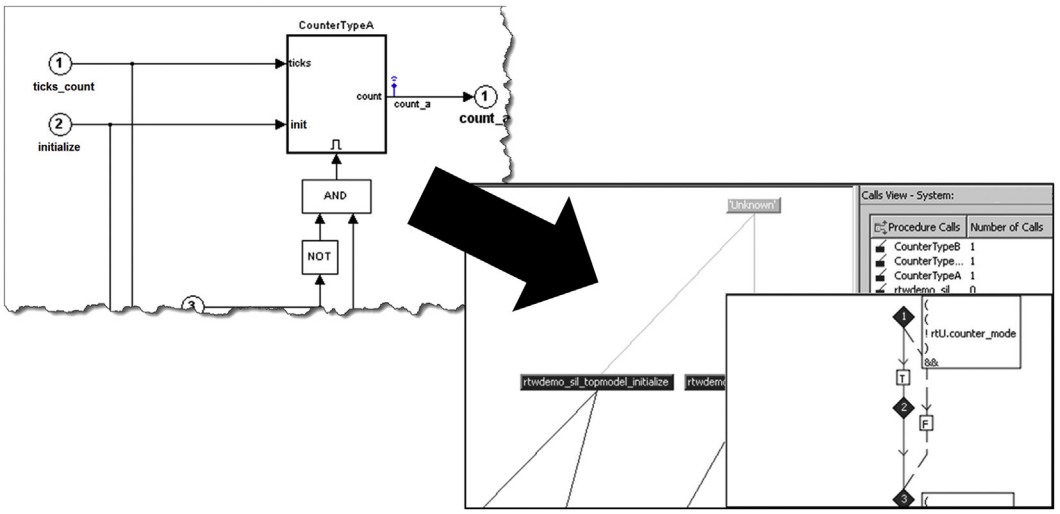


Fig. 28 Generating code coverage data in autogenerated code.

At first glance it may seem pointless testing autogenerated code at the source code level. However, this is not the case.

Even assuming that the code is not supplemented by manually generated code there is a multitude of problems that can exist in autogenerated code. For example, the conversion of floating-point arithmetic from the model simulation on the PC to floating-point arithmetic on the target may be erroneous and so require testing.

When a standard dictates that a particular level of code coverage is to be achieved it becomes necessary to demonstrate at source level (and conceivably at system level) that the code has indeed been executed.

It is sometimes in the nature of autogenerated code for redundant code to be generated, and many standards disallow such an inclusion. Static and dynamic analysis of the source code can reveal such superfluous additions and permit their removal.

There are also circumstances where the generated source code is expected to adhere to coding standards such as MISRA C:2012. The code generation suite may claim to meet such standards, but independent proof of that is frequently required.

8.2 Working With Legacy Code

Software test tools have been traditionally designed with the expectation that the code has been (or is being) designed and developed following a best practice development process.

Legacy code turns the ideal process on its head. Although such code is a valuable asset, it is likely to have been developed on an experimental, ad hoc basis by a series of "gurus"—experts who pride themselves at getting things done and in knowing the application itself, but

not necessarily expert at complying with modern development thinking and bored at having to provide complete documentation. That does not sit well with the requirements of such standards as DO-178C.

Frequently, this legacy software—often termed software of unknown pedigree (SOUP)—forms the basis of new developments. The resulting challenges do not just come from extended functionality. Such developments may need to meet modern coding standards and deploy updated target hardware and development tool chains, meaning that even unchanged functionality cannot be assumed to be proven.

The need to leverage the value of SOUP presents its own set of unique challenges.

8.2.1 The Dangers of Software of Unknown Pedigree

Many SOUP projects will initially have been subjected only to functional system testing, leaving many code paths unexercised and leading to costly in-service corrections. Even in the field it is highly likely that the circumstances required to exercise much of the code have never occurred 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, previously unexercised code paths are likely to be called into use by combinations of data never previously encountered (Fig. 29).

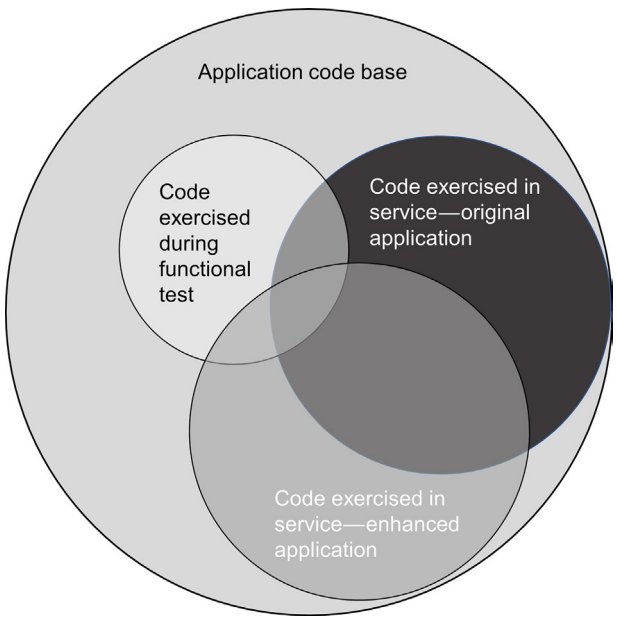


Fig. 29 Code exercised both on-site and by functional testing is likely to include many unproven execution paths. Code enhancements are prone to exercising previously unused paths even in established parts of the system.

The same commercial pressures that rule out a rewrite are likely to rule out the use of all the following options. As ever, they can be used selectively depending on the criticality of an application or its subsections.

8.2.2 Static and Dynamic Analysis of Software of Unknown Pedigree

In the enhancement of SOUP the existing code frequently defines the functionality of the system rather than documentation. In enhancing the code it is therefore vital that the functionality is not unintentionally modified. And, even where all source code remains identical, a new compiler or target hardware can introduce unintentional functionality changes with potentially disastrous results.

The challenge is to identify the building blocks within the test tools that can be used in an appropriate sequence to aid the efficient enhancement of SOUP.

There are five major considerations.

8.2.2.1 Improving the Level of Understanding

The system visualization facilities provided by many modern test tools are extremely powerful. Static call graphs provide a hierarchical illustration of the application and system entities, and static flow graphs show the control flow across program blocks.

Such call graphs and flow graphs are just part of the benefit of comprehensive analysis of all parameters and data objects used in the code. This information is particularly vital to enabling the affected procedures and data structures to be isolated and understood when work begins on enhancing functionality.

8.2.2.2 Enforcing New Standards

When new developments are based on existing SOUP it is likely that standards will have been enhanced in the intervening period. Code review analysis can highlight contravening code.

It may be that the enforcement of an internationally recognized coding standard 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 that could simply be less demanding or that could, for instance, place particular focus on portability issues.

The enforcement of new standards to legacy code is covered in more detail in the coding standards case study discussed earlier.

8.2.2.3 Ensuring Adequate Code Coverage

As previously established, code proven in service has often effectively been subjected only to extensive “functional testing” and may include many previously unexercised and unproven paths.

Structural coverage analysis addresses this issue by testing equally across the sources assuming each path through them has an equal chance of being exercised.

Although not offering a complete solution, system-wide functional testing exercises many paths and so provides a logical place to start.

Commonly a test tool may take a copy of the code under test and implant additional procedure calls ("instrumentation") to identify the paths exercised during execution. Textual code coverage reports are then often complemented with colored graphs to give insight into the code tested and into the nature of data required to ensure additional coverage.

Manually constructed unit tests can be used to ensure that each part of the code functions correctly in isolation. However, the time and skill involved in constructing a harness to allow the code to compile can be considerable.

The more sophisticated unit test tools minimize that overhead by automatically constructing the harness code within a GUI environment and providing details of the input and output data variables to which the user may assign values. The result can then be exercised on either the host or target machine.

To complement system test it is possible to apply code instrumentation to these unit tests and hence exercise those parts of the code that have yet to be proven. This is equally true of code that is inaccessible under normal circumstances such as exception handlers.

Sequences of these test cases can be stored, and they can be automatically exercised regularly to ensure that ongoing development does not adversely affect proven functionality, or to reestablish correct functionality when problems arise in service.

8.2.2.4 Dealing With Compromised Modularity

In some SOUP applications, structure and modularity may have suffered challenging the notion of testing functional or structural subsections of that code.

However, many unit test tools can be very flexible, and the harness code that is constructed to drive test cases can often be configured to include as much of the source code base as necessary. The ability to do that may be sufficient to suit a purpose.

If a longer term goal exists to improve overall software quality, then using instrumented code can help to understand which execution paths are taken when different input parameters are passed into a procedure—either in isolation or in the broader context of its calling tree.

8.2.2.5 Ensuring Correct Functionality

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, to the compiler or the target hardware, or to the data handled by the application.

Even with the aid of test tools, generating unit tests for the whole code base may involve more work than the budget will accommodate. However, the primary aim here is not to check that each procedure behaves in a particular way; it is to ensure that there have been no inadvertent changes to functionality.

By statically analyzing the code, test tools provide significant assistance for the generation of test cases, and the more sophisticated tools on the market are able to fully automate this process for some test case types. This assistance, whether partially or fully automated, will help to exercise a high percentage of the control flow paths through the code. Depending on the capabilities of the tool in use, input and output data may also be generated through fully or partially automated means. These data may then be retained for future use.

The most significant future use of these retained data will be in the application of regression tests, the primary function of which is to ensure that when those same tests are run on the code under development there are no unexpected changes. These regression tests provide the cross-reference back to the functionality of the original source code and form one of the primary benefits of the unit test process as a whole. As such the more feature rich of the available unit test tools will often boost the efficiency and throughput of regression tests via the ability to support batch processing.

8.3 Tracing Requirements Through to Object Code Verification (OCV)

With applications whose failure has critical consequences—people's lives could be at risk or there could be significant commercial impact—there is a growing recognition that stopping requirements traceability short of object code raises unanswered questions. There is an implied reliance on the faithful adherence of compiled object code to the intentions expressed by the author of the source code.

Where an industry standard is enforced a development team will usually adhere only to the parts of the standard that are relevant to their application—including OCV. And yet, OCV is designed to ensure that critical parts of an application are not compromised by the object code, which is surely a desirable outcome for any software whatever its purpose.

8.3.1 Industry Standards and Software Certification

Irrespective of the industry and the maturity of its safety standards the case for software that has been proven and certified to be reliable through standards compliance and requirements traceability is becoming ever more compelling.

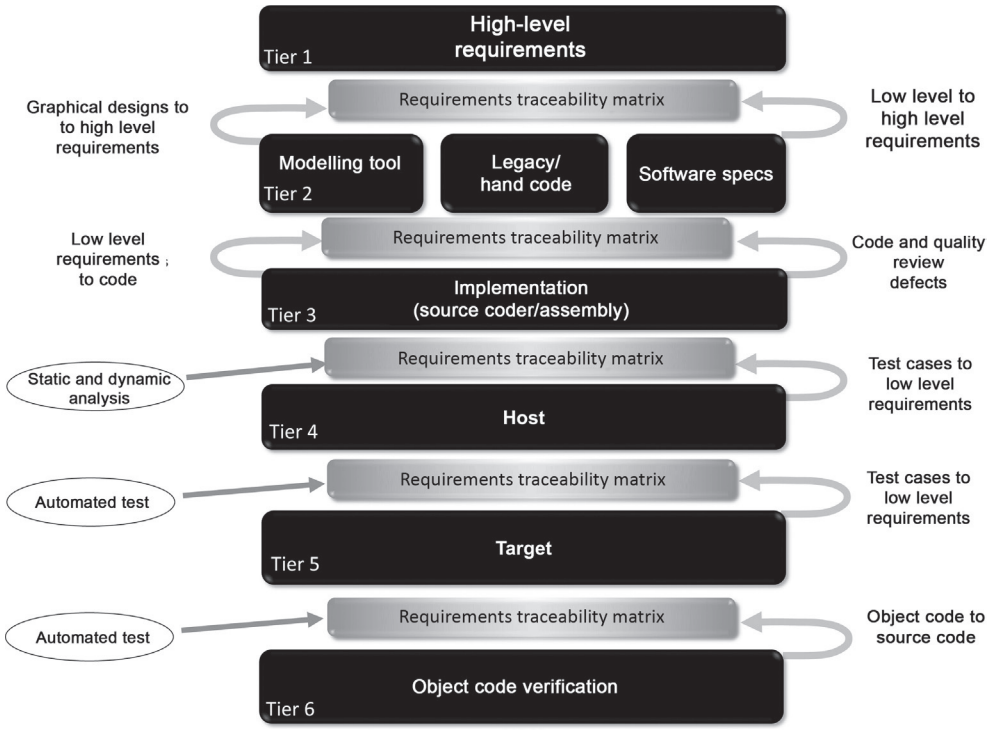


Fig. 30 The requirements traceability matrix (RTM) can be extended through to object code verification at the sixth tier.

When the RTM becomes the center of the development process it impacts all stages of design from high-level requirements through to target-based deployment. The addition of Tier 6 takes the target-based work a stage further, to tie in the comparison of the object and source code as part of the RTM and an extension to it (Fig. 30).

8.3.2 Object Code Verification (OCV)

So what is object code verification? The relevant section of the aerospace DO-178C standard describes the technique as follows:

"Structural coverage analysis may be performed on the Source Code, object code, or Executable Object Code. Independent of the code form on which the structural coverage analysis is performed, if the software level is A and a compiler, linker, or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish the correctness of such generated code sequences."

OCV therefore hinges on how much the control flow structure of the compiler-generated object code differs from that of the application source code from which it was derived.

8.3.3 Object Code Control Flow vs Source Code Control Flow

It is useful to illustrate this variation. Consider the following very simple source code:

```
void f_while4( int f_while4_input1, int f_while4_input2 )
{

   int f_while4_local1, f_while4_local2 ;

   f_while4_local1 = f_while4_input1 ;
   f_while4_local2 = f_while4_input2 ;

   while( f_while4_local1 < 1 || f_while4_local2 > 1 )
     {
        f_while4_local1 ++ ;
        f_while4_local2 -- ;
     }
}
```

This C code can be demonstrated to achieve 100% source code coverage by means of a single call thus:

```
f_while4(0,3);
```

and can be reformatted to a single operation per line like so:

```
1  void

1    f_while4 (
1    int f_while4_input1 ,
1    int f_while4_input2 )
1    {
1      int
1        f_while4_local1 ,
1        f_while4_local2 ;
1      f_while4_local1 = f_while4_input1 ;
1      f_while4_local2 = f_while4_input2 ;
-------------------------------------------------------------
2      while
2        (
2        f_while4_local1 < 1
2        ||
-------------------------------------------------------------
3        f_while4_local2 > 1
-------------------------------------------------------------
4        )
-------------------------------------------------------------
5        {
5          f_while4_local1 ++ ;
```

```
5           f_while4_local2 -- ;
5         }
-------------------------------------------------------------
6     }
```

The prefix for each of these reformatted lines of code identifies a "basic block"—that is, a sequence of straight line code. The resulting flow graph for the function shows both the structure of the source code and the coverage attained by such a test case with the basic blocks identified on the flowchart nodes (Fig. 31).

In this sequence of illustrations exercised parts of the code are shown using black nodes and solid branch lines.

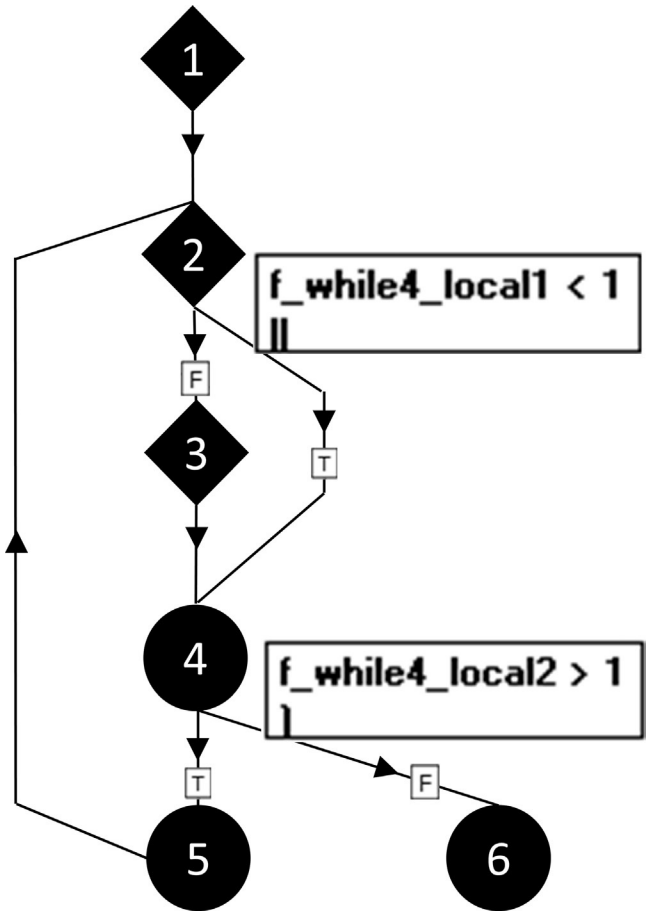


Fig. 31 A dynamic flowgraph showing that all source code has been exercised through a single function call.

The object code generated by a compiler will depend on the optimization setting, the compiler vendor, the target, and a host of other issues. The following shows just one example of resulting (reformatted) assembler code generated by a widely used commercially available compiler with optimization disabled.

It is not necessary to understand all the code to grasp the principle, but it is useful to be aware that the *ble* and *bgt* branch instructions ("branch less than equal" and "branch greater than," respectively) are used to divert control flow to their associated labels (L3 and L5). These branches lie within code that otherwise executes sequentially:

```
39 _f_while4:
40              push      fp
41              ldiu      sp,fp
42              addi      2,sp
43              ldi       *-fp(2),r0      ; |40|
44              ldiu      *-fp(3),r1      ; |41|
45              sti       r0,*+fp(1)      ; |40|
46              sti       r1,*+fp(2)      ; |41|
47              ble       L3              ; |43|    New test 2
48 ;*      Branch Occurs to L3            ; |43|
49              ldiu      r1,r0
50              cmpi      1,r0            ; |43|
51              ble       L5              ; |43|
52 ;*      Branch Occurs to L5            ; |43|    New test 3
53
54 L3:
55              ldiu      1,r0            ; |45|
56              ldiu      1,r1            ; |46|
57              addi      *+fp(1),r0      ; |45|
58              subri     *+fp(2),r1      ; |46|
59              sti       r0,*+fp(1)      ; |45|
60              cmpi      0,r0            ; |43|
61              sti       r1,*+fp(2)      ; |46|
62              ble       L3              ; |43|    New test 1
63 ;*      Branch Occurs to L3            ; |43|
64              ldiu      r1,r0
65              cmpi      1,r0            ; |43|
66              bgt       L3              ; |43|
67 ;*      Branch Occurs to L3            ; |43|
68
69 L5:
70              ldiu      *-fp(1),r1
71              bud       r1
```

It should be emphasized that there is NOTHING WRONG with this compiler or the assembler code it has generated. There are significant differences between the available constructs in high-level languages

(such as C and C++) and object code. To interpret the source code a compiler and/or linker therefore have to generate code that is not directly traceable to source code statements.

Consequently, the flowgraph looks different for the assembler code—and, in particular, using the identical test case generates a quite different flowgraph both in terms of appearance and importantly in terms of coverage.

This phenomenon is acknowledged in the DO-178C standard used in the aerospace industry. For the most safety-critical level A code it states that "if … a compiler, linker or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish the correctness of such generated code sequences." (Fig. 32).

It is clear from the flowchart and the assembler code that more tests are necessary to achieve 100% code coverage:

- New test 1. Line 62. End of block 3. Branch to L3.
  This ble branch always evaluates to false with the existing test data because it only exercises the loop once, and so only one of the two possible outcomes results from the test to see whether to continue. Adding a new test case to ensure a second pass around that loop exercises both true and false cases. A suitable example can be provided thus:

```
f_while4(-1,3);
```

- New test 2. Line 47. End of block 1. Branch to L3.
  This code contains an "or" statement in the "while" loop conditions. The existing test cases both result in the code:

```
f_while4_local1 < 1
```

  returning a "true" value.
  The addition of a new test case to return a "false" value will address that:

```
f_while4(3,3);
```

- New test 3. Line 52. End of block 2. Branch to L5.
  The remaining unexercised branch is the result of the fact that if neither of the initial conditions in the "while" statement is satisfied then the code within the loop is bypassed altogether via the ble branch.
  So, the final test added will provide such a circumstance:

```
f_while4(3,0);
```

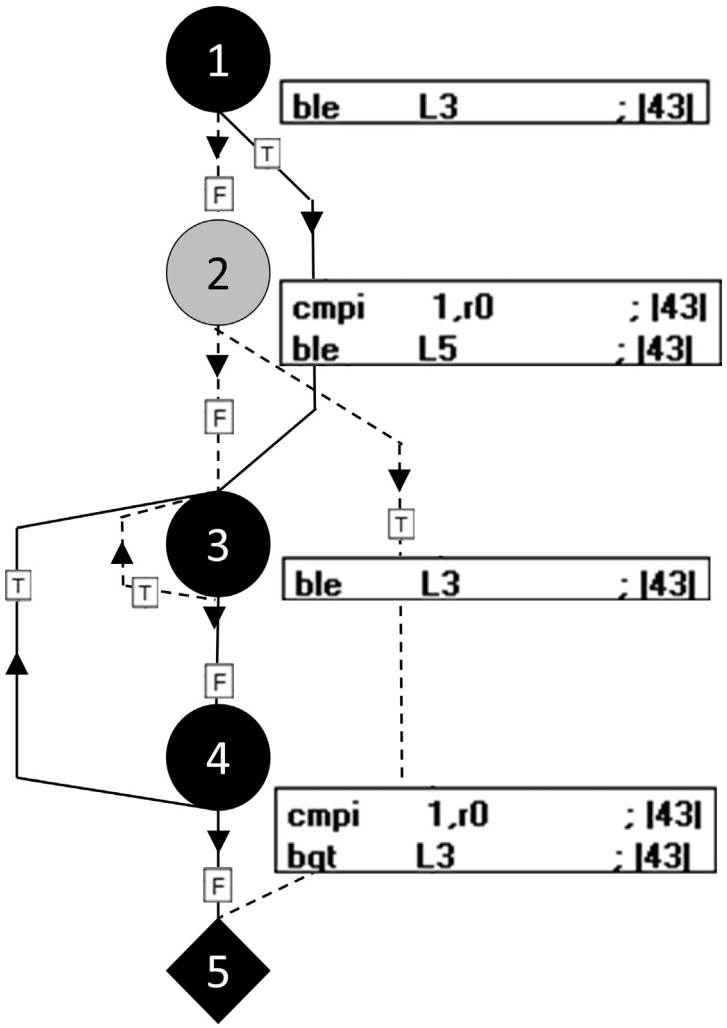


Fig. 32 A dynamic flowgraph showing the assembler code exercised through a single function call.

These three additional tests result in 100% statement and branch coverage of the assembler code (Fig. 33).

- So, to achieve 100% coverage of the assembler code, four tests are required:

```
f_while4(0,3);
f_while4(-1,3);
f_while4(3,3);
f_while4(3,0);
```

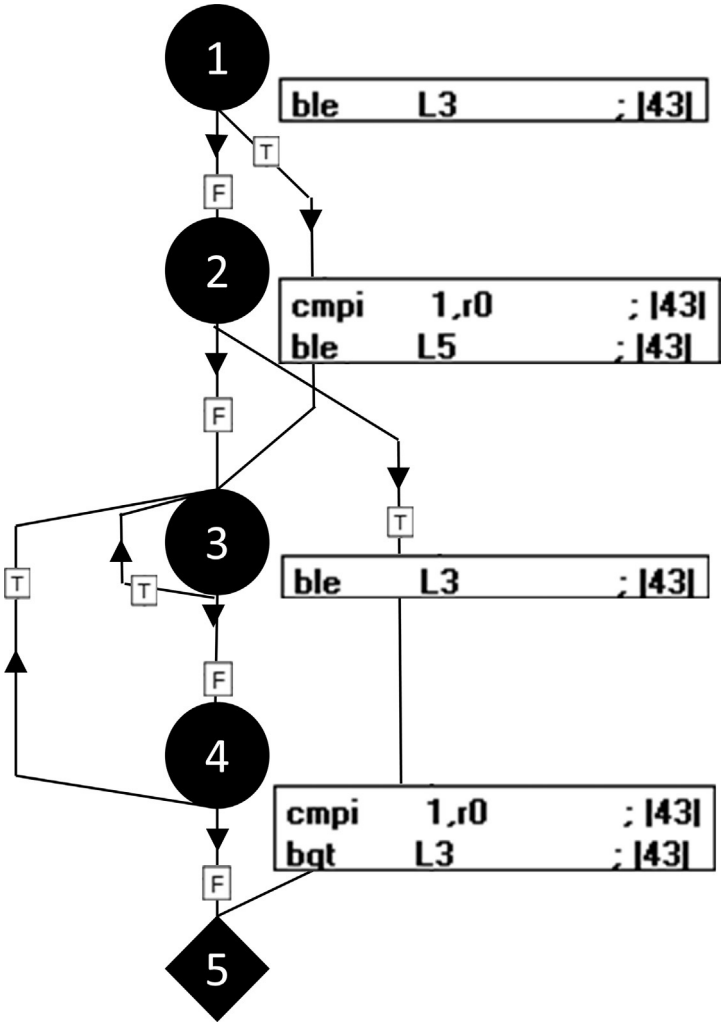


Fig. 33 A dynamic flowgraph showing 100% assembler code exercised through additional function calls.

8.3.3.1 Extending Source Code Coverage to Object Code Verification

If the principle of structural coverage analysis is justified, then it follows that object code verification (OCV) is also worthy of consideration.

In the general case structural coverage analysis provides evidence that ALL of the code base has been exercised. Such an approach has been proven to reduce the risk of failure and consequently is specified in most, if not all, industrial standards concerned with safety.

Structural coverage analysis offers a proven mechanism for ensuring that software is robust and safe. However, we have already established that merely exercising all of the source code does NOT prove that all of the object code has been similarly exercised and proven.

True, it is less likely that an unexercised route through the object code will lead to system failure, but even lesser risks can be unacceptable if a system is sufficiently safety, commercially, or mission critical.

In short, how big are the risks?

Further, consider the fact that our example mismatch between source and object code flowcharts was generated in a compiler with optimization disabled. Many more differences are likely as a result of compiler interpretation and optimization. While traditional structural coverage techniques are applied at the source code level, object code executes on the processor—and that is what really matters.

Any differences in control flow structure between the two can make for significant and unacceptable gaps in the testing process.

In some industries these gaps are acknowledged and accounted for. For example, in aerospace the DO-178B standard requires developers to implement OCV facilities for those elements of the application that have a Level A (safety-critical) classification. While this is often a subset of the application as a whole it has traditionally represented a significant amount of testing effort and hence has always required considerable resources.

Opportunities to implement automated, compiler-independent processes can help to reduce overall development costs by considerable margins, and conversely make the technique of OCV commercially justifiable in other fields.

8.3.3.2 Automated Object Code Verification

Automated OCV solutions can provide a complete structural coverage analysis solution for both source and object code from unit to system and integration levels.

Typical solutions combine both high- and object-level (assembler) code analysis tools, with the object-level tool variant being determined by the target processor that the application is required to run on. A typical example might see C/C++ and PowerPC Assembler analysis tools teamed together to provide the required coverage metrics.

8.3.3.3 Object Code Verification at the Unit Level

Tools are available that enable users to create test cases for structural coverage of high-level source and apply these exact same test cases to the structural coverage of the corresponding object code.

A driver program is generated by such a unit test tool which encapsulates the entire test environment, defining, running, and monitoring the test cases through initial test verification and then subsequent

regression analysis. When used for OCV this driver may be linked with either the high-level source unit or the associated object code. In so doing users can apply a uniform test process and compare code to determine any discrepancies or deficiencies.

If any such structural coverage discrepancies or deficiencies are identified at the object level, users are then presented with an opportunity to define additional test cases to close any gaps in the test process. The obvious advantage of identifying and applying corrective action at such an early development stage is that it is much easier and cheaper. It also significantly increases the quality of the code and the overall test process with the latter reaping benefits at the later stages of integration and system testing and then onward in the form of reduced failure rates/maintenance costs when the application is in the field.

While the code is still under development, together with satisfying the necessary OCV requirements in a highly automated and cost-effective manner developers can also benefit from the considerable additional test feedback. The results of these analysis facilities can be fed back to the development team with the possibility that further code and design deficiencies may be identified and rectified, further enhancing the quality of the application as a whole.

8.3.3.4 Justifying the Expense

It is clear that OCV has always involved significant overhead and that even in the aerospace sector it is only enforced as a requirement for the most demanding safety integrity levels. Even then, the elements nominated for object code verification in these applications usually represent a subset of the application as a whole—a specialist niche indeed.

However, there is precedence for this situation. Until quite recently unit test has been considered by many as a textbook nicety for the purposes of the aircraft and nuclear industry. More recently it has found a place in automotive, railway, and medical applications, and now the ever-increasing capabilities and ease of use of automated unit test tools has introduced a commercial justification of such techniques even when risks are lower.

Most applications include key elements in the software: a subset of code that is particularly critical to the success of the application and can be identified in the application requirements. The software requiring OCV can be identified and traced through an extension to the RTM.

The advent of tools to automate the whole of that process from requirements traceability right through to OCV challenges the notion that the overhead involved can only justify the technique in very rare circumstances. Just as for unit test before it perhaps the time has come for OCV to be commercially justifiable in a much broader range of circumstances.

9 Implementing a Test Solution Environment

9.1 Pragmatic Considerations

Like so many other things in business life, ultimately the budget that is to be afforded to the test environment depends on commercial justification. If the project under consideration has to be shown to comply with standards in order to sell, then that justification is straightforward. It is much less clear-cut if it is based entirely on cost savings and enhanced reputation resulting from fewer recalls.

Although vendors make presentations assuming developers are to work on a virgin project where they can pick and choose what they like, that is often not the case. Many development projects enhance legacy code, interface to existing applications, are subject to the development methods of client organizations and their contractual obligations, or are restricted by time and budget.

The underlying direction of the organization for future projects also influences choices:

- Perhaps there is a need for a quick fix for a problem project in the field, or a software test tool that will resolve a mystery and an occasional runtime error crash in final test.
- Maybe there is a development on the order books which involves legacy code requiring a one-off change for an age-old client, but which is unlikely to be used beyond that.
- Perhaps existing legacy code cannot be rewritten, but there is a desire and mandate to raise the quality of software development on an ongoing basis for new developments and/or the existing code base.
- Or perhaps there is a new project to consider, but the lessons of problems in the past suggest that ongoing enhancement of the software development process would be beneficial.

To address a particular situation it is initially useful to consider how each of the five key attributes discussed earlier fit into the development process.

9.2 Considering the Alternatives

Given that vendors are generally not keen to highlight where their own offering falls short some insight into how to reach such a decision would surely be useful.

Fig. 34 superimposes the different analysis techniques on a traditional “V” development model. Obviously, a particular project may use another development model. In truth the analysis is model agnostic and a similar representation could be conceived for any other development process model—waterfall, iterative, agile, etc.

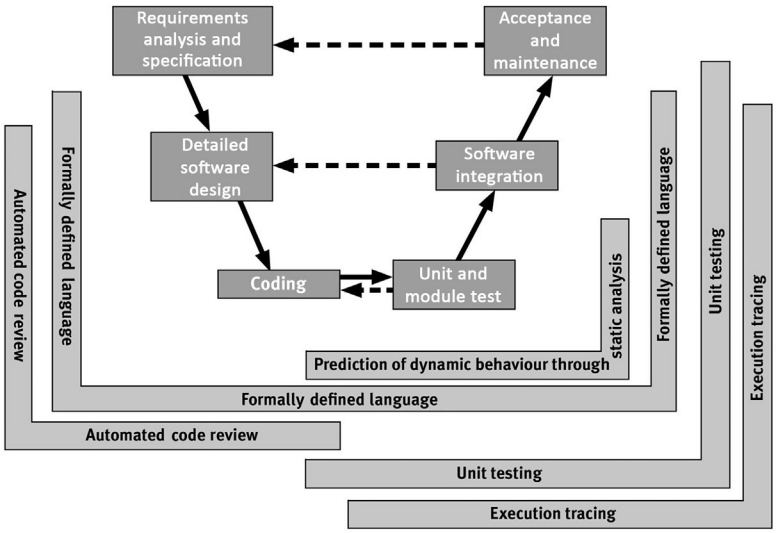


Fig. 34 The five fundamental test tool attributes directly relate to the specific development stages of design, code, test and verification, etc.

The extent to which it is desirable to cover all elements of the development cycle depends very much on the initial state of development and the desired outcome.

Each of the five key test tool attributes has merit.

There is a sound argument supporting traditional formal methods, but the development overheads for such an approach and the difficulty involved in applying it retrospectively to existing code limits its usefulness to the highly safety-critical market.

Automated code review checks for adherence to coding standards and is likely to be useful in almost all development environments.

Of the remaining approaches, dynamic analysis techniques provide a test environment much more representative of the final application than static predictions of dynamic analysis as well as the means to provide functional testing.

Where requirements traceability is key within a managed and controlled development environment the progressive nature of automated code review followed by unit, integration, and system test aligns well within the overall tiered concept of most modern standards. It also fulfills the frequent requirement or recommendation to exercise the code in its target environment.

Where robustness testing is considered desirable and justified it can be provided by means of the automatic definition of unit test vectors, or through the use of the static prediction of dynamic behavior. Each of these techniques has its own merits, with the former exercising code in its target environment and the latter providing a means

to exercise the full data set rather than discrete test vectors. Where budgetary constraints permit, these mutually exclusive benefits could justify the application of both techniques. Otherwise, the multifunctional nature of many of the available unit test tools makes them very cost-effective.

If there is a secondary desire to evolve corporate processes toward the current best practice, then both automated code review and dynamic analysis techniques have a key role to play in requirements management and traceability, with the latter being essential to show that the code meets its functional objectives.

If the aim is to find a pragmatic solution to cut down on the number of issues displayed by a problem application in the field, then each of the robustness techniques (i.e., the static analysis of dynamic behavior or the automatic definition of unit test vectors) has the potential to isolate tricky problems in an efficient manner.

9.2.1 When Is Unit Test Justifiable?—Case Study

It is perhaps useful to consider one of the five attributes to illustrate a possible thought process to be applied when deciding where to invest.

Unit testing cannot always be justified. Moreover, sometimes it remains possible to perform unit test from first principles without the aid of any test tool at all.

There are pragmatic judgments to be made.

Sometimes such a judgment is easy. If the software fails, what are the implications? Will anyone be killed, as might be the case in aircraft flight control? Will the commercial implications be disproportionately high, as exemplified by a continuous plastics production plant? Or are the costs of recall extremely high, perhaps in an automobile's engine controller? In these cases extensive unit testing is essential and hence any tools that may aid in that purpose make sense.

On the other hand, if software is developed purely for internal use or is perhaps a prototype, then the overhead in unit testing all but the most vital of procedures would be prohibitive.

As might be expected, there is a gray area. Suppose the application software controls a mechanical measuring machine where the quantity of the devices sold is low and the area served is localized. The question becomes: Would the occasional failure be more acceptable than the overhead of unit test?

In these circumstances it is useful to prioritize the parts of the software that are either critical or complex. If a software error leads to a strangely colored display or a need for an occasional reboot, it may be inconvenient but not in itself justification for unit test. On the other hand, the unit test of code that generates reports showing whether machined components are within tolerance may be vital. Hence, as

we have already seen advocated by leading standards such as DO-178B, significant benefit may be achieved through a decision to apply the rigor of unit test to a critical subset or subsets of the application code as a whole.

9.2.2 When Are Unit Test Tools Justifiable?

Again, it comes down to cost. The later a defect is found in the product development, the more costly it is to fix—a concept first established in 1975 with the publication of Brooks' *The Mythical Man-Month* and proven many times since through various studies.

The automation of any process changes the dynamic of commercial justification. That is especially true of test tools given that they make earlier unit test much more feasible. Consequently, modern unit test almost implies the use of such a tool unless only a handful of procedures are involved.

The primary function of such unit test tools is to assist with the generation and maintenance of the harness code that provides the main and associated calling functions or procedures (generically "procedures"), with the more sophisticated tools on the market being able to fully automate this process. The harness itself facilitates compilation and allows unit testing to take place.

The tools not only provide the harness itself, but also statically analyze the source code to provide the details of each input and output parameter or global variable in an easily understood form. Where unit testing is performed on an isolated snippet of code, stubbing of called procedures can be an important aspect of unit testing. This can also often be partially or fully automated to further enhance the efficiency of the approach.

High levels of automation afforded by modern unit test tools makes the assignment of values to the procedure under test a simple process and one that demands little knowledge of the code on the part of the test tool operator. This creates the necessary unit test objectivity because it divorces the test process from that of code development where circumstances require it and from a pragmatic perspective substantially lowers the level of skill required to develop unit tests.

It is this ease of use that means unit test can now be considered a viable arrow in the development quiver, targeting each procedure at the time of writing. When these early unit tests identify weak code it can be corrected while the original intent remains very fresh in the mind of the developer.

10 Summary and Conclusions

There are hundreds of textbooks about software test and many that deal with only a specialist facet of it. It is therefore clear that a chapter such as this cannot begin to cover the whole subject in detail.

Some elements of software test remain unmentioned here. What about testing for stack overflow? Timing considerations? And multi-threaded applications, with their potentially problematic race and lethal embrace conditions?

In preference to covering all such matters superficially the technique demonstrated in this chapter—of "drilling down" into a topic to shed sufficient light on its worth in a particular circumstance—is sound and will remain so even for test techniques as yet unavailable. It is therefore applicable in those matters not covered in any detail here.

In each case these are techniques that can be deployed or not as circumstances dictate; the whole genre of software test techniques and tools constitute a tool kit just as surely as a toolbox holding spanners, hammers, and screwdrivers.

And, just like those handyman's friends, sometimes it is possible to know from a superficial glance whether a technique or test tool is useful, while at other times it needs more detailed investigation.

The key then is to be sure that decisions to follow a particular path are based on sufficient knowledge. Take the time to investigate and be sure that the solution you are considering will prove to be the right one for a particular circumstance.

Consider the option of developing in-house tests, and when commercially marketed test tools are considered be sure to ask for an evaluation copy.

Choosing the wrong technique or the wrong tool can be a very costly and embarrassing mistake indeed.

Questions and Answers

Question 1 Why is it best to identify bugs as early as possible?

Answer 1 The earlier a bug is identified, the less it costs to fix it. Compare the cost of a developer spotting a bug in his automotive code as he is writing it with the cost of a vehicle recall when the software is in service!

Question 2 Identify the industrial sectors to which these functional safety standards apply:

- **a.** DO-178C
- **b.** ISO 26262
- **c.** IEC 62304

Answer 2

- **a.** DO-178C, "Software considerations in airborne systems and equipment certification," applies to the aerospace industry
- **b.** ISO 26262, "Road vehicles—Functional safety," applies to the automotive industry
- **c.** IEC 62304, "Medical device software—Software life cycle processes," applies to the medical device industry

Question 3	Why is the achievement of 100% branch coverage more demanding than the achievement of 100% statement coverage?
Answer 3	Where 100% branch coverage is achieved both false and true decisions will be executed for each branch (100% statement coverage is achievable without exercising both false and true decisions for each branch)
Question 4	Why does 100% source code coverage not imply 100% object code coverage?
Answer 4	There are significant differences between the available constructs in high-level languages (such as C and C++) and object code. To interpret the source code a compiler and/or linker therefore have to generate code that is not directly traceable to source code statements
Question 5	Are adherents to functional safety standards such as DO-178C, ISO 26262, IEC 61508, and IEC 62304 obliged to use software test tools?
Answer 5	There is no obligation for any developer or development team to use software test tools to comply with any of the functional safety standards. There is a commercial decision to be made in each case whether the efficiency gained from the application of tools justifies expenditure on them

Further Reading

[1] English Oxford Living Dictionaries, https://en.oxforddictionaries.com/definition/test.

[2] IEC 61508-1:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems.

[3] The Mythical Man-Month, Addison-Wesley, ISBN: 0-201-00650-2, 1975.

[4]. DO-178C, Software Considerations in Airborne Systems and Equipment Certification (December 13, 2011), RTCA.

[5] MISRA C:2012, Guidelines for the Use of the C Language in Critical Systems, March 2013. ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF).

[6] MISRA C++:2008 Guidelines for the Use of the C++ Language in Critical Systems (June 2008), ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF).

[7] Computability: Turing, Gödel, Church, and Beyond (January 30, 2015) The MIT Press, Paperback, B. Jack Copeland (Editor, Contributor), Carl J. Posy (Editor, Contributor), Oron Shagrir (Editor, Contributor), Martin Davis (Contributor) et al., ISBN 978-0262527484.

[8] ISO 26262:2011 Road vehicles—Functional safety.

[9] IEC 62304 Edition 1.12015–05 Medical device software—Software life cycle processes.

[10] The International Obfuscated C Code Contest, https://www.ioccc.org.

[11] Joint Strike Fighter Air Vehicle C++ Coding Standards for The System Development And Demonstration Program (December 2005), Document Number 2RDU00001 Rev. C, Lockheed Martin Corporation.

[12] Barr group Embedded C Coding Standard, https://barrgroup.com/Embedded-Systems/Books/Embedded-C-Coding-Standard.

[13] SEI CERT C Coding Standard, Rules for Developing Safe, Reliable, and Secure Systems, 2016. Edition.

[14] MISRA C:2012 AMD1, Additional security guidelines for MISRA C:2012, https://www.revolvy.com/page/Frances-E.-Allen.

[15] Just The Facts 101: Discovering Computers, 2010. Complete: First edition. Cram 101.
[16] Classics in Software Engineering, Yourdon Press, ISBN: 0-917072-14-6, 1979. https://cse.buffalo.edu/~rapaport/111F04/greatidea3.html.
[17] Software Reliability, Principles and Practices, G. J. Myers, Wiley, New York, 1976.
[18] Microsoft Excel, https://www.microsoft.com/en-gb/p/excel/cfq7ttc0k7dx?activetab=pivot%3aoverviewtab.
[19] High Integrity C++ Coding Standard Version 4.0, www.codingstandard.com.
[20] S. 882 (111th): Drug and Device Accountability Act of 2009, https://www.govtrack.us/congress/bills/111/s882.
[21] FDA Orders Recall Of Baxter Colleague Infusion Pumps (May 2010), Lucy Campbell, https://www.lawyersandsettlements.com/lawsuit/baxter-colleague-infusion-pumps-recall.html.
[22] DO-178B, Software Considerations in Airborne Systems and Equipment Certification (December 1992), EUROCAE.
[23] CENELEC–EN 50128, Railway applications—Communication, signaling and processing systems—Software for railway control and protection systems, 2011.
[24] IEC 61513, Nuclear power plants—Instrumentation and control important to safety—General requirements for systems, 2011.
[25] IEC 61511, Functional safety—Safety instrumented systems for the process industry sector, 2016.
[26] SAE J3061, Cybersecurity Guidebook for Cyber-Physical Vehicle Systems, 2016.
[27] ISO/SAE 21434, Scope (DRAFT) Road vehicles–Cybersecurity engineering, 2017.