SuperGuard: Qualifying the C Standard Library for use in safety-critical applications

Introduction

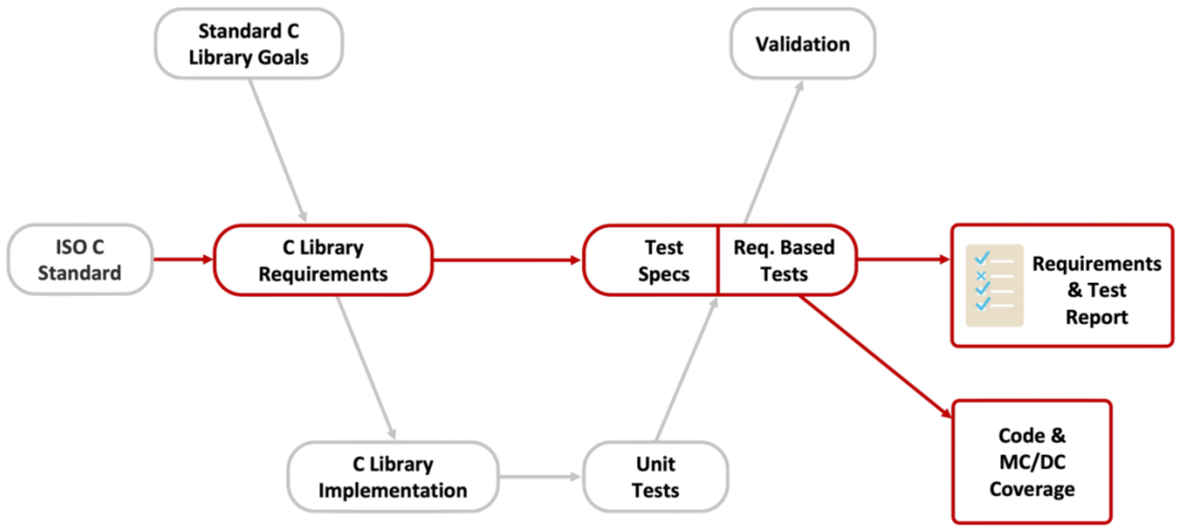
Software solutions play an ever-increasing role in safety-critical and safety-related systems, with the result that software malfunctions now represent liabilities and a real threat in terms of injury, loss of life, the interruption of essential services, or damage to the environment. As a result, international standards organizations such as the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have published widely recognized and widely adopted standards against which software developers can certify the safety of their software. Examples include ISO 26262 (*Road vehicles – Functional safety*) for automotive, EN 50128 (*Communication, signaling and processing systems - Software for railway control and protection systems*) for rail transport, and IEC 61508 (*Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems*) for industrial applications.

The responsibility for demonstrating that application software, and the software methods, processes, and toolchains used to develop it, comply with the relevant functional safety standards lies firmly with the application developer. However, it remains a fact that significant parts of the toolchain lie outside the developer's control. This is one of the reasons why compiler validation – an area in which Solid Sands is already a world leader – has become a key issue for developers of safety-critical systems. Virtually no compiler is bug-free, so it is extremely important to know where a compiler malfunctions so that compiler errors can be avoided.

It is also true that a significant part of the code that becomes part of the complete application is likely to be compiled with a different use-case, set of compiler options, and compilation environment from those being used by the developer. This is because part of the code that typically ends up in an application comprises pre-compiled library functions, such as those in the C Standard Library (libc) that is often supplied in binary format as part of a software development kit (SDK).

Contrary to the commonly held belief that because a library is supplied in binary format it is insensitive to any particular use-case – i.e. the code is invariant – in practice this is not the case. The inclusion of macros and type-generic templates frequently makes library components use-case sensitive. So even if the library was pre-qualified by the SDK supplier using the same compiler delivered with the SDK, the matching use-case, compiler options, and target hardware environment requirements are almost certain not to have been met, making it difficult to demonstrate functional safety standard compliance.

To overcome this limitation, Solid Sands has introduced a new library qualification tool called the SuperGuard C Library Safety Qualification Suite – a requirements-based test suite for the C Standard Library with full traceability from individual test results back to requirements derived from the ISO C language specification. SuperGuard can be used to support qualification of C Standard Library implementations for safety-critical applications both for unmodified third-party library implementations and self-developed or self-maintained implementations. Its role in the V-Model for software development is shown below.



The Aim of Qualification

Qualification of a software library is critical because code from the library is linked into the application and installed onto the target device. If a library component is defective, the functional safety of the entire application is therefore jeopardized. Every functional safety standard has its own specific objectives when it comes to the use of software libraries, but in general they all share a common goal: to verify that the library implementation complies to its specification. ISO 26262 provides two routes to library qualification, detailed separately in ISO 26262 Part 8 and ISO 26262 Part 6. The SuperGuard C Library Safety Qualification Suite can be used in both cases.

ISO 26262 Part 8, Clause 12: Qualification of Software Components

In terms of being characterized as a commercial off-the-shelf (COTS) product, libraries are covered in Part 8, Clause 12: 'Qualification of Software Components'. This clause addresses the need for qualification of existing software components such as libraries “to provide evidence for their suitability for re-use”. It specifically mentions “software libraries from third-party suppliers”, which clearly includes the standard libraries supplied with commercial SDKs. The clause also applies to re-used internal software and open-source software.

Part 8, Clause 12 states that a pre-requisite for qualification is a statement of the requirements of the software component. It also suggests that evidence that a software component complies with these requirements should “primarily be based on requirements-based tests”, that this can be achieved through the “application of a dedicated qualification test suite”, that it should “cover both normal operating conditions and behavior in the case of failure”, and that it should “display no known errors that may lead to a violation of safety requirements”.

Fortunately, for both the C and C++ languages, the library specification exists and is publicly accessible, so the starting point for rigorous requirements-based testing is already there. In fact, the existence of and the ability to test against both the language and the library specification is one of the reasons why the C and C++ programming languages are so ubiquitous in the developer community. Yet, the language specification is not written as a list of requirements. A key feature of Solid Sands’ SuperGuard test suite is that all its library tests are firmly based on requirements derived from the ISO C language definition.

To meet the requirement that the test suite should cover both normal and failure conditions, it should exercise each function both within its boundary conditions and outside its boundary conditions and verify the function’s error handling. The requirements derived from the ISO C specification to create SuperGuard include verification of the required failure behavior defined in the specification.

Reliable evidence of there being no known errors that could lead to a violation of safety requirements not only relies on effective requirements-based testing, it also relies on structural code coverage for Automotive Safety Integrity Level D (ASIL D) — the highest integrity level for automotive applications. To ensure evidence of completeness, SuperGuard provides high structural code coverage and high Modified Condition/Decision Coverage (MC/DC).

SuperGuard also includes the analysis and testing of equivalence classes and boundary values, and error guessing based on the best available knowledge and experience of a library function’s behavior.

ISO 26262 Part 6: Product Development at the Software Level

ISO 26262 Part 6 adopts a parallel approach, treating library code in the same way as all other application code that runs on the target. Ironically, it does not mention libraries as a separate category of code at all, which may be the reason why the need for library qualification is often overlooked.

Qualification of the C Standard Library as detailed in ISO 26262 Part 6 is more involved than the approach detailed in Part 8 because Part 6 addresses all phases of software development.

Part 6, Table 7: 'Methods for software unit verification' includes requirements-based testing as one of its recommendations for all ASIL levels. As described above for 'qualification of software components' (ISO 26262 Part 8, Clause 12), SuperGuard comprehensively supports requirements-based testing, although in practice it is recommended that developers use more than one of the methods listed in the table to verify a C Standard Library implementation.

SuperGuard also employs all the methods listed in Part 6, Table 8: 'Methods for deriving test cases for software unit testing'. Method 1a, 'Analysis of requirements' is used to break down the C Standard Library specification into testable requirements. The C Standard Library description is a mix of (sometimes implicit) definitions, restrictions on the use of functions, and definitions of function behavior. In SuperGuard, these have also been translated into testable requirements that form the basis of the test suite, including requirements for the handling of anomalous cases defined in the language standard.

Methods 1b, 'Generation and analysis of equivalence classes', and 1c, 'Analysis of boundary values', relate to the partitioning of the input value domains of the functions and are addressed in SuperGuard's test specifications, which provide the link between the requirements and the tests.

Method 1d, 'Error guessing based on knowledge or experience' is based on Solid Sands' knowledge of difficult implementation areas for the C library, and on the collection of regression tests that have been collected since the initial development of Solid Sands' SuperTest compiler test and verification suite more than 30 years ago.

In this approach, SuperGuard plays a substantial role on the right-hand side of the V-Model for software development. In relation to ISO 26262 Part 6, Clause 9: 'Software unit verification', it addresses the following:

- Compliance of the standard C library implementation to its requirements
- Verification of the hardware-software interface by running the tests on the target hardware
- Confidence in the absence of unintended functionalities by the verification of failure cases and by monitoring code coverage
- Verification of resource requirements by running the tests on the target hardware

How SuperGuard Tests are Developed

When implementing the requirements-based testing recommended in Part 8 and Part 6 of the ISO 26262 functional safety standard, the main issue with the C and C++ Standard Library specifications is that although they provide a detailed behavioral description for each function, neither of them defines a clear set of requirements. The necessary requirements for each function must therefore be created from the behavioral descriptions.

The SuperGuard C Library Safety Qualification Suite incorporates the proven test suite for the C Standard Library already included in Solid Sands' world-leading SuperTest compiler test and verification suite, which has tracked the (ISO) language specifications for more than 30 years. However, SuperGuard goes much further than SuperTest in terms of its reporting capabilities, documenting requirements, individual tests and test results in accordance with functional safety standards such as ISO 26262, EN 50128 and IEC 61508.

The tests in SuperGuard's test suite are designed according to the following principles, making them suitable for a wide range of development environments.

SuperGuard tests are behavioral tests — i.e. they verify that the behavior of the implementation complies with the library specification. Each test executes the construct or function under test and compares the results of execution with the expected ('model') results defined in the library specification. The test itself reports success or failure to the test driver.

In order to check the behavior of the implementation, these tests are compiled and executed in an execution environment, which means that the entire toolchain, including the target processor, is involved in each test. This makes SuperGuard suitable for hardware-in-the-loop verification of the library.

The tests for the freestanding part of the library (typically used in bare-metal systems) require minimal resources. Most SuperGuard tests can run on systems with less than 4K memory, making it possible to use SuperGuard on very small embedded systems.

To implement requirements-based testing, SuperGuard provides a detailed breakdown of the C Standard Library specifications into testable implementation requirements together with test specifications describing how each requirement is tested. By linking individual test execution results back to the corresponding test specification, test requirement, and standard library function, SuperGuard provides the full traceability needed for requirements-based testing. To provide evidence of completeness, it provides close to 100% structural code coverage for more than 80% of the functions, with high Modified Condition/Decision Coverage (MC/DC). Note that this addresses the library implementation itself, not the underlying OS layer.

An Example

Each library test in the SuperGuard suite is developed according to a consistent methodology, illustrated by the **strncpy** function shown below. This is the specification in Section 7.21.2.4 of the C99 language definition:

7.21.2.4 The strncpy function

Synopsis

1 ***#include <string.h>***
char * strncpy(char * restrict s1, const char * restrict s2, size_t n);

Description

2 *The* ***strncpy*** *function copies not more than* ***n*** *characters (characters that follow a null character are not copied) from the array pointed to by* ***s2*** *to the array pointed to by* ***s1****. If copying takes place between objects that overlap, the behavior is undefined.*

3 *If the array pointed to by* ***s2*** *is a string that is shorter than* ***n*** *characters, null characters are appended to the copy in the array pointed to by* ***s1****, until* ***n*** *characters in all have been written.*

Returns

4 *The* ***strncpy*** *function returns the value of* ***s1****.*

The most curious point about this specification is that Paragraph 2 specifies no lower bound of characters for **strncpy()** to copy. It reads: "*copies not more than* ***n*** *characters*." At no point does the specification require that any characters are copied from **s2** to **s1**. In Paragraph 3 it does state an action, namely that **s1** is padded with null characters. Taken literary, a correct implementation according to this specification would be to just write **n** null characters to **s1**.

Yet, this is not what the function is meant to do, nor what it is expected to do. The general understanding is that the function does copy as many as possible characters from **s2** to **s1** until either the string **s2** or **n** is exhausted. There is no confusion about that and it appears nobody complained about this phrasing since ANSI C89 because the same wording is still present in C18. But, to define requirements we have to be a bit more precise.

The first step in our test development process is to extract a set of requirements (REQs) from this description, taking into account what the function is actually supposed to do. These are:

REQ-copystring: *If* ***s2*** *points to a string with a length '****l2****' (as it is defined by* ***strlen()****) that is less than* ***n****,* ***strncpy()*** *shall copy* ***l2*** *characters, in order, from the array* ***s2*** *into the array* ***s1****.*

REQ-copyn: *If* ***s2*** *does not point to a string with length less than* ***n****,* ***strncpy()*** *shall copy the first* ***n*** *characters, in order, from the array* ***s2*** *into the array* ***s1****.*

REQ-shorter: *If **s2** points to a string with a length that is less than **n**, **strncpy()** shall append null characters (‘**\0**’) after the copied characters in array **s1** until **n** characters in total have been written.*

REQ-nomore: ***strncpy()** shall not write into the target array **s1** beyond the first **n** characters.*

REQ-nochange: ***strncpy()** shall not modify array **s2**.*

REQ-return: ***strncpy()** shall return the value of **s1**.*

The requirement **REQ-nochange** follows from the declaration of **s2** as a **const**ant array, but the declaration itself does not guarantee that an implementation of **strncpy()** does not write to **s2**.

For each of these requirements, a test specification is then developed. The test specification defines how a test verifies that the requirement is true. A single test specification usually leads to a number of different test cases that cover the input and output domains of the function. The test cases are implemented by the test. The test specification links the requirement to the tests.

For example, the test specifications for the **REQ-copystring** and **REQ-nomore** requirements are as follows:

Test specification for **REQ-copystring**: *Call the **strncpy()** function with different values for the **n** parameter (including **n**==0) equal to and larger than the length of the origin string. Verify that the origin string is copied into the target array up to the terminating null character.*

Test specification for **REQ-nomore**: *For all test cases, verify that the character with index **n** in the target array **s1** is not modified. If that fits with the test, verify that also no characters beyond **n** are modified.*

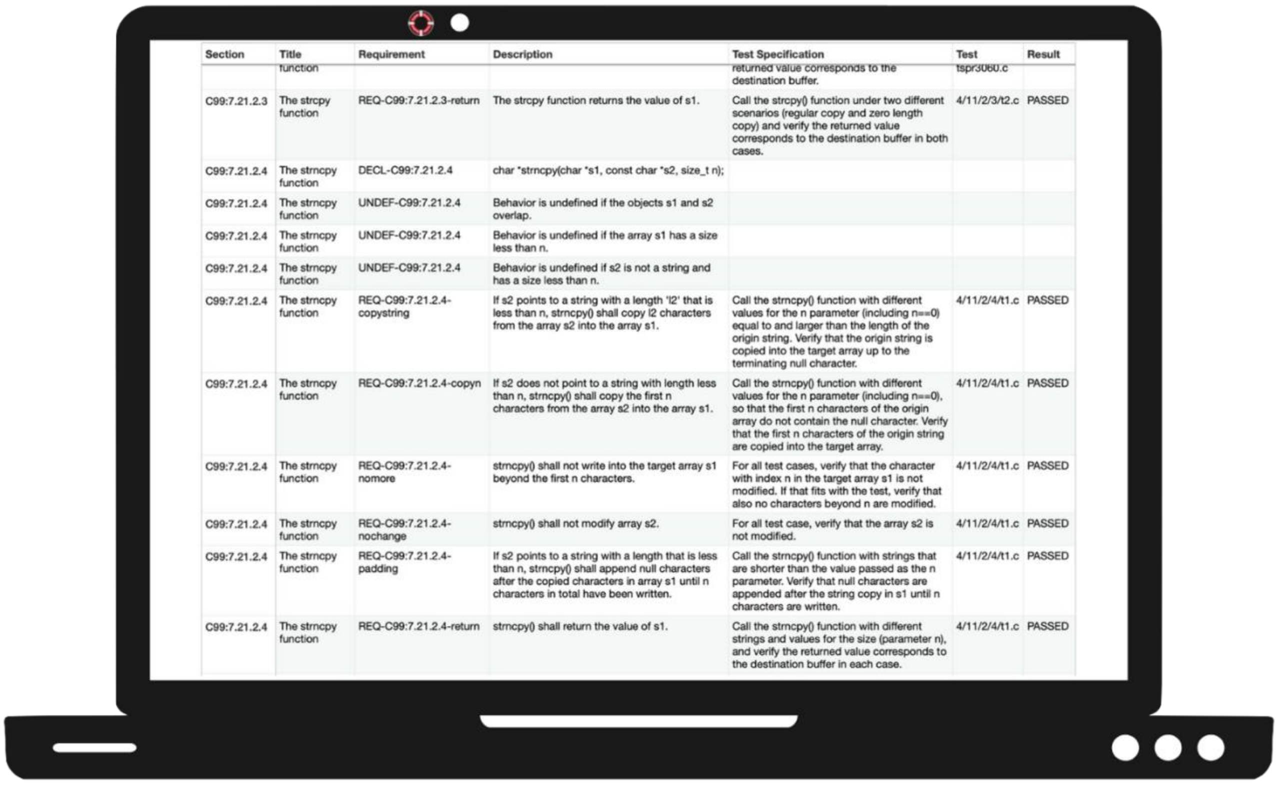
In this case, the test specification **REQ-nomore** is implemented in the same test file as the other tests cases for **strncpy()**. Since the requirement must unconditionally hold for every call to **strncpy()** anyway, instead of creating new tests for this test specification, it is implemented by simply piggybacking an additional check on every case test for the other requirements instead of creating new tests for it.

Dealing With Header Files and Function-like Macros

The C language throws in one more spanner to complicate the tester’s life. Which is that not all functions in the C Standard Library are only implemented as pre-compiled binaries. Many are also heavily dependent on information contained in source header files.

These header files, which define things such as types, global variables, and macros, are as much a part of the library as the (pre-compiled) library functions. Many functions are implemented both as a real function and as a macro, and for speed and efficiency it is common practice to use the macro implementation. Both are tested by SuperGuard.

Unlike the corresponding binaries, function-like macros are not pre-compiled. They are compiled by the SDK's compiler together with the application source code. It is therefore important that, together with other content in the header files, they are verified for the specific use-case of a given safety-critical application. In the C++ library, the use of macros is elevated to an even higher level through the use of type-generic templates that only exist in the headers.



Section	Title	Requirement	Description	Test Specification	Test	Result
	function			returned value corresponds to the destination buffer.	tspr3060.c	
C99:7.21.2.3	The strcpy function	REQ-C99:7.21.2.3-return	The strcpy function returns the value of s1.	Call the strcpy() function under two different scenarios (regular copy and zero length copy) and verify the returned value corresponds to the destination buffer in both cases.	4/11/2/3/t2.c	PASSED
C99:7.21.2.4	The strncpy function	DECL-C99:7.21.2.4	char *strncpy(char *s1, const char *s2, size_t n);			
C99:7.21.2.4	The strncpy function	UNDEF-C99:7.21.2.4	Behavior is undefined if the objects s1 and s2 overlap.			
C99:7.21.2.4	The strncpy function	UNDEF-C99:7.21.2.4	Behavior is undefined if the array s1 has a size less than n.			
C99:7.21.2.4	The strncpy function	UNDEF-C99:7.21.2.4	Behavior is undefined if s2 is not a string and has a size less than n.			
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-copystring	If s2 points to a string with a length 'l2' that is less than n, strncpy() shall copy l2 characters from the array s2 into the array s1.	Call the strncpy() function with different values for the n parameter (including n==0) equal to and larger than the length of the origin string. Verify that the origin string is copied into the target array up to the terminating null character.	4/11/2/4/t1.c	PASSED
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-copyn	If s2 does not point to a string with length less than n, strncpy() shall copy the first n characters from the array s2 into the array s1.	Call the strncpy() function with different values for the n parameter (including n==0), so that the first n characters of the origin array do not contain the null character. Verify that the first n characters of the origin string are copied into the target array.	4/11/2/4/t1.c	PASSED
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-nomore	strncpy() shall not write into the target array s1 beyond the first n characters.	For all test cases, verify that the character with index n in the target array s1 is not modified. If that fits with the test, verify that also no characters beyond n are modified.	4/11/2/4/t1.c	PASSED
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-nochange	strncpy() shall not modify array s2.	For all test case, verify that the array s2 is not modified.	4/11/2/4/t1.c	PASSED
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-padding	If s2 points to a string with a length that is less than n, strncpy() shall append null characters after the copied characters in array s1 until n characters in total have been written.	Call the strncpy() function with strings that are shorter than the value passed as the n parameter. Verify that null characters are appended after the string copy in s1 until n characters are written.	4/11/2/4/t1.c	PASSED
C99:7.21.2.4	The strncpy function	REQ-C99:7.21.2.4-return	strncpy() shall return the value of s1.	Call the strncpy() function with different strings and values for the size (parameter n), and verify the returned value corresponds to the destination buffer in each case.	4/11/2/4/t1.c	PASSED

Where Requirements-based Testing Fits With ISO 26262 Part 8 Clause 12

Through these requirements-based test methodologies, SuperGuard ensures availability of the following key elements set out in ISO 26262: Part 8, Clause 12.4.1 for a software component to be considered as qualified:

12.4.1 a) the specification of the software component

The specification of the C and C++ Standard Libraries are based on publicly available ISO standards, with SuperGuard adding clear functional requirements to the behavioral descriptions contained in these standards.

12.4.1 b) evidence that the software component complies with its requirements

By translating the functional requirements extracted from the library specification's behavioral descriptions into test specifications and test designs and assembling the resultant tests into a comprehensive executable test suite, SuperGuard provides evidence that an implementation of the C Standard Library complies with the extracted requirements and hence the behavioral description.

12.4.1 c) evidence that the software component is suitable for its intended use

When SuperGuard is used to verify a C Standard Library implementation, it creates a detailed report of the PASS/FAIL status of all tests, with full traceability back to the functional requirements extracted from the specification's behavioral descriptions.

SuperGuard also meets the requirements of ISO 26262 Part 8, Clause 12, paragraphs 12.4.2.2, 12.4.2.3 and 12.4.2.4, which are referred to in 12.4.1b above. Clause 12.4.2.2 indicates that verification of a software component can be done using requirements-based testing and a dedicated test suite, which is what SuperGuard provides. It also states that the verification must "*cover both normal operating conditions and behavior in the case of failure*" and "*display no known errors that may lead to a violation of safety requirements allocated to this software component*". SuperGuard tests the functional requirements for all defined failure behaviors stated in the ISO C language standard.
Clause 12.4.2.3 defines an additional requirement for the use of software components in ASIL-D applications, stating that structural coverage must be measured in order to evaluate the completeness of the test cases.

Clause 12.4.2.4 states that the verification process detailed in Clause 12 can only be applied to an unchanged implementation of the software component. While some developers replace library functions by their own specialized implementations, these modifications normally only apply to a few functions. Since the C Standard Library is highly modular and virtually all function implementations are independent of the rest of the library, the majority of library functions can still be verified under the provisions of Part 8, Clause 12, allowing the application developer to focus on qualifying only those functions that have been modified using the provisions detailed in Part 6.

Code Coverage Analysis

During construction of the SuperGuard test suite, special focus was placed on the code coverage achieved for a mature and popular open-source C Standard

Library implementation to meet the ASIL D requirement of Clause 12.4.2.3. For many functions in that library, SuperGuard achieves 100% coverage, in addition to high MC/DC coverage. The areas where SuperGuard has lower coverage are typically related to implementation-defined behavior, for which no requirements can be derived from the language specification.

Although every library implementation is different, ultimately they all have to handle a similar case analysis. This means that SuperGuard's high code coverage benefits code coverage for all C Standard Library implementations.

Anomalous Cases

There are two ways SuperGuard tests handle anomalous cases. The first relates to defined behavior resulting from an anomalous input — for example, passing a negative number to the function **sqrt**() must return the value **NaN** (assuming IEC 60559 arithmetic). Although this is an anomalous case, the behavior of the function is fully defined and can be verified like any other behavior.

The second relates to requirements that can be verified by the compiler. For example, if a function must have a **void** return type, a test can try to use the return value with the expectation that it will generate a compiler error. Such a test is called an *x-test* in SuperGuard and the filename of these tests starts with an *x*. X-Tests PASS if the compiler raises an error at compilation and FAIL if it does not. X-Tests are never executed.

Summary and Conclusions

Safety-critical applications require software developers to do everything in their power to ensure that their development processes, toolchains, and application code pose no risk of injury, loss of life, the interruption of essential services, or damage to the environment. When using third-party and/or commercial off-the-shelf (COTS) tools and components, such as compilers and standard libraries, developers should not assume that these tools and components are error-free, or that pre-qualification implies that to be the case. Qualification is only truly valid if it is carried out in precisely the same development environment and under exactly the same use-case scenario as used in the application.

Employing the same library test suite included in Solid Sands' SuperTest compiler test and verification solution, its SuperGuard C Library Safety Qualification Suite adds the traceability needed to relate the results of requirements-based tests – the recommended method of testing in functional safety standards such as ISO 26262 – back to the C Standard Library specification. Full traceability is provided by breaking down the ISO C Standard Library functional specifications into clearly defined requirements, developing suitable test specifications to check those requirements and implementing them in accordance with ISO 26262 recommendations. In addition, it allows software developers to perform these tests in the same development environment, under the same use-case conditions, and on the same target hardware used in their application, with close to 100% structural code coverage. By generating a comprehensive qualification report tailored to the needs of ISO 26262 certification organizations, SuperGuard alleviates much of the burden of demonstrating the integrity of library components used in safety-critical applications.