



Library Qualification From Requirement to Test Specification to Test



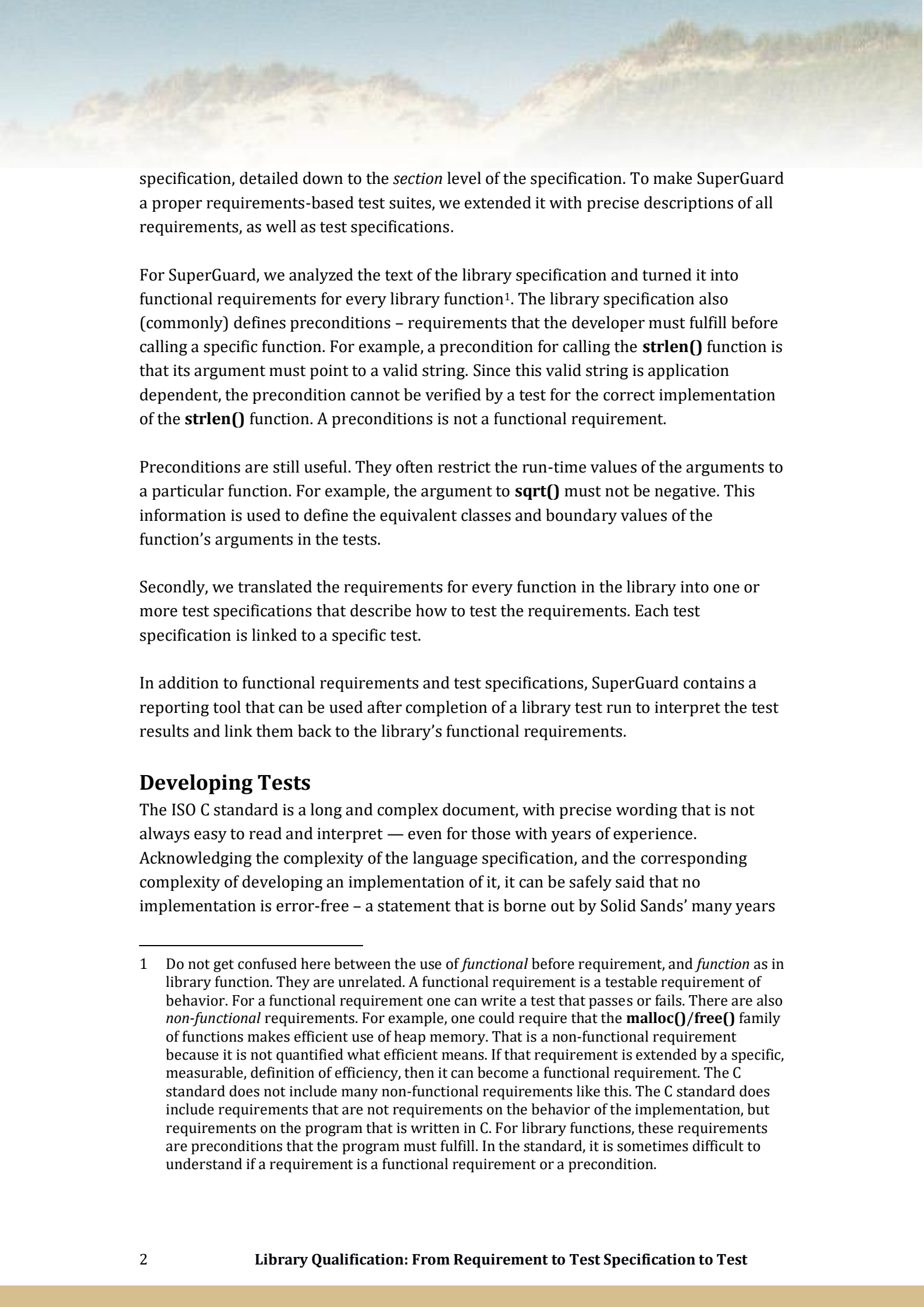
If functions from the C standard library are used in a safety-critical application, ISO 26262 requires their verification. For a library that is developed or modified in-house, ISO 26262-6 Clause 9 applies (which is about verification of application software in general). If the library is from an external source, the qualification method described in ISO 26262-8 Clause 12 can be used (for software originally developed in another project, commercial off-the-shelf software, and even open source).

In either case, 'requirements-based testing' is necessary to verify the C standard library implementation. The tests used must be developed using a combination of techniques: *analysis of the requirements; the use of equivalence classes; the definition of boundary values; and 'error guessing'*. At Solid Sands we prefer to call the last of these techniques 'experience', rather than 'error guessing'.

The starting point for developing a requirements-based test suite for the C standard library is the library specification in the ISO C language standard. This specification describes the behavior of library functions from the perspective of the library user. It does not simply list the functional requirements. In this document we will discuss how you get from this user-level behavior document to a set of functional requirements, and then to a test suite that matches the requirements of ISO 26262 and other safety standards.

Functional Requirements and Test Specifications

In our test suites, tests are organized according to the ISO standard C library



specification, detailed down to the *section* level of the specification. To make SuperGuard a proper requirements-based test suites, we extended it with precise descriptions of all requirements, as well as test specifications.

For SuperGuard, we analyzed the text of the library specification and turned it into functional requirements for every library function¹. The library specification also (commonly) defines preconditions – requirements that the developer must fulfill before calling a specific function. For example, a precondition for calling the **strlen()** function is that its argument must point to a valid string. Since this valid string is application dependent, the precondition cannot be verified by a test for the correct implementation of the **strlen()** function. A preconditions is not a functional requirement.

Preconditions are still useful. They often restrict the run-time values of the arguments to a particular function. For example, the argument to **sqrt()** must not be negative. This information is used to define the equivalent classes and boundary values of the function’s arguments in the tests.

Secondly, we translated the requirements for every function in the library into one or more test specifications that describe how to test the requirements. Each test specification is linked to a specific test.

In addition to functional requirements and test specifications, SuperGuard contains a reporting tool that can be used after completion of a library test run to interpret the test results and link them back to the library’s functional requirements.

Developing Tests

The ISO C standard is a long and complex document, with precise wording that is not always easy to read and interpret — even for those with years of experience.

Acknowledging the complexity of the language specification, and the corresponding complexity of developing an implementation of it, it can be safely said that no implementation is error-free – a statement that is borne out by Solid Sands’ many years

1 Do not get confused here between the use of *functional* before requirement, and *function* as in library function. They are unrelated. A functional requirement is a testable requirement of behavior. For a functional requirement one can write a test that passes or fails. There are also *non-functional* requirements. For example, one could require that the **malloc()/free()** family of functions makes efficient use of heap memory. That is a non-functional requirement because it is not quantified what efficient means. If that requirement is extended by a specific, measurable, definition of efficiency, then it can become a functional requirement. The C standard does not include many non-functional requirements like this. The C standard does include requirements that are not requirements on the behavior of the implementation, but requirements on the program that is written in C. For library functions, these requirements are preconditions that the program must fulfill. In the standard, it is sometimes difficult to understand if a requirement is a functional requirement or a precondition.

of test experience. All implementations should therefore be thoroughly tested in order to find bugs. But what should a test look like? Here is a simplified test for a requirement from C's library section:

```
#include <assert.h>
#include <stdio.h>

int main( void ){
    switch( BUFSIZ ){
        case BUFSIZ:
            assert( 1 );
        default:
            assert( 0 );
    }
    return 0;
}
```

The C standard requires that the **BUFSIZ** macro “*expands to an integral constant expression*” (C90:7.9.1). The test uses a property of the **switch** statement to test that **BUFSIZ** is indeed an integral constant. Regarding the **switch** statement, the standard states that “*the expression of each case label shall be an integral constant expression*” (C90:6.6.4.2). If an implementation successfully compiles and executes the above code, it verifies that our requirement on **BUFSIZ** is met.

The previous test is a positive test – i.e. a test that must be compiled and executed successfully. There is also another kind of test, called a negative test, that contains incorrect code – for example, a constraint violation or a construct that is not allowed by the standard. The following is an example of a negative test.

```
#include <stdlib.h>

int main( void ){
    sizeof( free( NULL ) );
    return 0;
}
```

According to the C standard, the return type of the **free()** function is **void** (i.e. “*The free() function returns no value*” (C90:7.10.3.2)). Taking into account that the **void** return type is an incomplete type (“*The void type comprises an empty set of values, it is an incomplete type that cannot be completed*” (C90:6.1.2.5)), a useful test implementation is to call the **free()** function wherever an incomplete type, such as **void**, is not allowed.

For this we can use a property of the **sizeof** operator, since “*The sizeof operator shall not be applied to an expression that has [...] an incomplete type [...]*” (C90:6.3.3.4).

This test must not be compiled successfully, and the compiler must issue a diagnostic in order to pass the test. If the compiler produces an executable program from this source code, there is an error in the implementation.

Unlike these two examples, requirements cannot always be turned into tests, because sometimes there is insufficient information in the library specification. This happens for features referred to as *implementation defined*, for which part of the specification is left to the implementation. For example, the C specification allows for many different implementations of the 'locale' feature that are not defined in the C specification. Thus, for this requirement of the **strftime()** function: “*The conversion specifier B is replaced by the locale's full month name*” (C90:7.12.3.5), it is not possible to create a test based on the C specification alone.

Library Tests

Although it is the compiler that turns the source code into executable code, the two previous tests are aimed at testing requirements from the library section of the language standard (**stdio.h** and **stdlib.h**, respectively). The compiler is a tool, but the library is software that actually ends up in a target device. That is why library testing is so important and, when talking about safety, why the qualification process for libraries is more elaborate than for compilers. Here is another example, complete with requirement and test specification. Let's say that we extract a requirement (named *REQ-C90:7.9.6.1-evaluate* in SuperGuard) from the C standard regarding the number of arguments in the **fprintf()** function:

“*If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) ...*” (C90:7.9.6.1).

How can we create a test to verify this requirement? A possible approach is to open a file in writing mode, call the **fprintf()** function to write a string on it, and place a last extra argument with a side-effect – a post-incremented counter – that has no corresponding conversion specifier in the format string of the call. In that way, checking the value of the counter after the call is sufficient to verify that the argument is evaluated. This is the test code:



```
#include <assert.h>
#include <stdio.h>

int main( void ){
    int count = 0;
    FILE *stream = fopen( "cval01.dat", "w" );

    assert( stream != NULL );
    fprintf( stream, "%s", "Hello", count++ );
    fclose( stream );
    assert( count == 1 );
    return 0;
}
```

And the explanation of how we built the test is the test specification for this requirement:

```
/* TEST SPEC REQ-C90:7.9.6.1-evaluate
   Create a file for writing and print a string to it
   using the fprintf() function. Place an extra argument
   in the call, which has a side effect: a post-incre-
   mented counter. After the call, verify that the counter
   is modified, even if there is no corresponding conversion
   for it specified in the format string.
*/
```

You could rightly argue that the evaluation of arguments is a property of the language more than the specific **fprintf()** function. However, the requirement is explicitly mentioned for **fprintf()**, which is a good reason to verify it in the library test suite. There is a second reason for verifying this requirement that is based on experience (see ‘error guessing’ above). The **printf()** family of functions is often optimized by compilers. If the format and arguments of these functions are such that they can be simplified (as is the case above), compilers often replace the call with a simpler form. In these **printf()** inspired optimizations, the requirement must still be met.

In Summary

Considering the complexity of the C language specification and the importance of its correct implementation, standard library implementations cannot be taken for granted. The best way to verify that your library is implemented correctly is to qualify it with a test suite that includes both the requirements extracted from the standard and test specifications that fulfill those requirements.

(c) Copyright 2022 by Solid Sands B.V., Amsterdam, The Netherlands
SuperTest™ and SuperGuard™ are trademarks of Solid Sands B.V., Amsterdam, The Netherlands