Library Qualification

From Requirements to Test Designs

If functions from the C standard library are used in a safety-critical application, ISO 26262 requires verification of the library. If the library is developed or modified in-house, ISO 26262-6 Clause 9 applies (which is about Software Unit Verification for 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 supplied by third-party suppliers, commercial off-the-shelf software).

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*'. Based on 30+ years of regression test development for SuperTest™, at Solid Sands we prefer to call the last of these techniques 'experience', rather than 'error guessing'.

The starting point for developing SuperTest's 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 implementation requirements. How we get from this user-level behavior document to a set of implementation requirements, and then to a test suite that matches the requirements of ISO 26262 and other safety standards, is the subject of this document.

Implementation Requirements and Test Designs

In the SuperTest Library Suites, tests are organized according to the ISO standard C library specification, with granularity down to the 'Section' level of the specification. The specification texts are not broken down into requirements. To bridge the gap between the specification and individual tests, the SuperTest C Library Suite can be extended with a fine-grained add-on, a safety package.

To create this safety package, we first analyzed the text of the library specification and turned it into implementation requirements for every library function. Note the qualifier '*implementation'*. The library specification also defines function pre-conditions – requirements that the programmer/application must fulfill before calling a specific function. For example, a pre-condition for calling the **strlen()** function is that its argument must point to a valid string. However, because this valid string is application dependent, the pre-condition cannot be verified by a test for correct implementation of the **strlen()** function. Such pre-conditions are therefore not implementation requirements.

That is not to say that pre-conditions are not used by SuperTest's tests. Pre-conditions often define limits on 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 each function's implementation requirements into one or more test designs that describe how to test the requirements. Each test design is linked to a specific test. For examples, see below.

In addition to these implementation requirements and test designs, the safety package 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 implementation requirements.

Creating 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 expertise. In fact, it is not uncommon for the language committee itself to publish fixes and modifications (the so-called Defect Reports: C11, C2X). That is how it should be. Because without doing so the committee would not be able to accomplish its task, which is to specify a language definition (syntax, semantics, constraints, library, etc.) as clearly and unambiguously as possible, both for compiler developers and programmers.

Acknowledging the complexity of the language specification, and the corresponding complexity of developing a language implementation, it can be safely said that no implementation is error-free (GCC, CLANG) – a statement that is borne out by Solid Sands' many years 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 code sample involving 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). To test that **BUFSIZ** is indeed an integral constant, the test uses a property of the **switch** statement. 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. (Of course, SuperTest also has tests to verify the statement about the **switch** statement itself.)

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

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 good *test design* for an X-Test 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). That test looks like this:

```
#include <stdlib.h>

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

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

Note that, unlike in these two cases, not all requirements can 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 a requirement that could be extracted from the **strftime()** function such as "*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 that checks all the existing 'locales'.

Library Tests

Although it is the compiler that turns the source code into object/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 the 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. For example, let's say that we extract a requirement (named *REQ-C90:7.9.6.1-evaluate* in the safety package) 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) but are otherwise ignored*" (C90:7.9.6.1).

How can we create a test to verify this requirement? A possible approach could be 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", "SuperTest is the Best", count++ );
   fclose( stream );

   assert( count == 1 );
   return 0;
}
```

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

```
/*  TEST DESIGN REQ-C90:7.9.6.1-evaluate:
   Create a file for writing and print a string on it using
   the fprintf() function. Place a last extra argument in
   the call, a post-incremented counter, in order to verify
   that it is evaluated even if there is no corresponding
   conversion specified for it 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 for making sure. There is a second reason for verifying this requirement, one 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 designs that fulfill those requirements.

Solid Sands B.V. / Postbus 7897 / 1008 AB Amsterdam / The Netherlands
Phone + 31 20 244 0199 / KvK 61176249
www.solidsands.nl