機能安全に求められるライブラリの認定

Library Qualification From Requirements to Test Designs

セーフティクリティカルなアプリケーションで C 標準ライブラリの関数を使用する場合、ISO 26262 ではそのライブラリを検証することが必要です。社内で開発・変更されるライブラリには、ISO 26262 Part6 第 9 節が適用されます（これはアプリケーションソフトウェア一般に対するソフトウェアユニット検証に関するものです）。もしライブラリが外部のソースである場合、ISO 26262 Part8 第 12 節に記載されている認定方法を使用することができます（サードパーティサプライヤーから提供されたソフトウェアや COTS ソフトウェアの場合）。

いずれの場合も、C 標準ライブラリの実装を検証するには、「要件ベースのテスト」が必要です。使用されるテストは、*要件の分析*、*同値クラスの使用*、*境界値の定義*、そして「*エラー推測*」などの手法を組み合わせて開発する必要があります。Solid Sands では 30 年以上の SuperTest™ のリグレッションテスト開発に基づいて、この手法の最後のものを「エラー推測」ではなく「経験」と呼んでいます。

SuperTest の C標準ライブラリに対する要件ベースのテストスイートを開発するには、ISO C言語規格におけるライブラリ仕様を出発点としています。これはライブラリを使用するユーザーの観点からライブラリ関数の振舞いを記述したもので、単に実装の要件を列挙したものではありません。このユーザーレベルでの振舞いのドキュメントから、実装要件の集合を、そして ISO 26262やその他の安全規格に合致するテストスイートをどのようにして得るのか、が本論文の主題です。

実装要件とテスト設計

SuperTest のライブラリスイートは、ISO 規格の C ライブラリ仕様に従ってテストが構成されており、仕様の「セクション」のレベルまで詳細化されています。仕様の文章は要件に細分されてはいないので、仕様と個々のテストの間のギャップを埋めるため、SuperTestのCライブラリスイートは、安全パッケージと称するきめ細かいアドオンで拡張しました。

この安全パッケージを作成するため、まず、各ライブラリ関数に対してその仕様の文章を分析し、実装要件に変えました。「*実装*」という言葉に注意してください。ライブラリの仕様では関数の事前条件（その関数を呼び出す前にプログラマやアプリケーションが満たさなければならない要件）も定義しています。たとえば、`strlen()`関数を呼び出す事前条件は、その実引数が有効な文字列を指していなければならないというものです。しかし、有効な文字列というものはアプリケーションに依存するので、`strlen()`関数の正しい実装を確認するテストでは、この事前条件を検証することはできません。したがって、このような事前条件は実装要件ではありません。

とはいえ事前条件が SuperTest のテストで使用されていないというわけではありません。通常、事前条件は個別の関数の実引数に対する制限を定義します。たとえば、`sqrt()`の実引数は負であっ

てはなりません。この情報は、テストにおいて関数の実引数の同値クラスと境界値を定義するために使用されます。

次に、各関数の実装要件を、その要件のテスト方法を記述した一つ以上のテスト設計に変換しました。各テスト設計は特定のテストにリンクしています。以下にいくつか例を示しています。

これらの実装要件とテスト設計に加えて、安全パッケージには、ライブラリのテスト実行が完了した後にテスト結果を解釈し、それらをライブラリ実装の要件にリンクするために使用するレポートツールが含まれています。

テストの作成

ISO の C 言語規格は長くて複雑な文書で、長年の専門知識を持つ人であっても容易には解釈できない細かい表現が使われています。実際、この言語規格委員会自身が修正や変更すること（いわゆる不具合報告：C11 ではここ、C2X ではここ）も珍しくありません。当然そうあるべきで、そうしなければ、委員会はコンパイラ開発者とプログラマ双方のために、言語定義（構文、意味、制約、ライブラリなど）を可能な限り明確かつ明白に規定するという任務を達成することができないからです。

言語仕様の複雑さと、それに応じて言語実装が複雑であることを認めると、エラーのない実装は存在しないといっても過言ではありません（GCC のバグリスト、CLANG のバグリスト）。Solid Sands の長年のテスト経験がそれを裏付けています。したがって、バグを見つけるためにはすべての実装を徹底的にテストする必要があります。しかし、テストとはどのようなものでしょうか？ 以下は、C 言語のライブラリセクションの要件を含む簡略化したコードサンプルです。

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

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

C 言語規格では、`BUFSIZ` マクロは「*整数定数式に展開する*」必要があります(C90:7.9.1)。`BUFSIZ` が実際に整数定数であることをテストするために、このテストでは `switch` 文の性質を使用しています。`switch` 文に関して規格では「*各case ラベルの式は，整数定数式でなければならず*」と規定されています(C90:6.6.4.2)。実装において上記のコードを正しくコンパイルして実行する場合、`BUFSIZ` に対する要件が満たされていることを確認します。(もちろん、SuperTest には `switch` 文自体に関して文を検証するテストもあります)

このようなテストは、*ポジティブテスト*（T テスト）すなわち、コンパイルして正常に実行されることが必要なテストです。また、ネガティブテスト（X テスト）と呼ばれる別の種類のテストがあり、

これは、正しくないコード（制約の違反や規格で許可されていない要素など）が含まれているものです。Xテストの例を以下に示しています。

C 言語規格では `free()`関数の戻り値の型は `void` です（すなわち「*free 関数は，値を返さない*」（C90:7.10.3.2））。`void`戻り値の型が不完全型であること（「*void型の値の集合は空とする。それは，完全にすることのできない不完全型とする*」（C90:6.1.2.5））を考慮すると、Xテストに対する優れたテスト設計は、`void`のような不完全型が許可されていない場合には、`free()`関数を呼び出すものです。このために、「*sizeof 演算子は，… 不完全型をもつ式，… に対して適用してはならない*」（C90:6.3.3.4）ので、`sizeof` 演算子の性質を使用することができます。テストは次のようになります。

```
#include <stdlib.h>

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

このテストはコンパイルに成功してはならず、コンパイラツールはテストをパスするために診断メッセージを発行しなければなりません。コンパイラがこのソースコードから実行可能プログラムを生成する場合、そのコンパイラの実装にエラーがあるということです。

上述の二つの場合とは異なり、すべての要件がテストに変換できるわけではありません。ライブラリ仕様には不十分な情報があるからです。それは「処理系定義」と呼ばれ、実装に任せられる機能に対して起こります。たとえば、Cの仕様では「ロケール」機能は定義されておらず、多くの異なる実装が可能です。したがって、`strftime()`関数から抽出できうる要件－「（変換指定子）*%B は，ロケールの簡略化されていない月の名前に置き換わる*」（C90:7.12.3.5）などでは、既存の「ロケール」すべてをチェックするテストを作成することはできません。

ライブラリテスト

ソースコードをオブジェクトや実行コードに変換するのはコンパイラですが、上記二つのテストは言語規格（それぞれ `stdio.h` と `stdlib.h`）のライブラリのセクションからの要件をテストすることを目的としています。コンパイラはツールであり、ライブラリは実際にターゲットデバイスに配置されるソフトウェアです。そのことが、ライブラリのテストが非常に重要であって、安全性について話すときライブラリの認定プロセスがコンパイラに対するものよりも複雑である理由です。たとえば、`fprintf()`関数の実引数の数に関する C の規格から次の要件（安全パッケージの *REQ-C90:7.9.6.1-evaluate*という名前）を抽出するとします。

「*実引数が残っているにもかかわらず書式が尽きてしまう場合，余分の実引数は評価するだけで無視する*」（C90:7.9.6.1）。

この要件を検証するテストを作成するにはどうすればよいでしょうか。可能なアプローチとして、ファイルを書き込みモードで開き、`fprintf()`関数を呼び出してそこに文字列を書き込み、最後に副作用（ポストインクリメントのカウンタ）をもつ余分な実引数を置き、呼び出しの書式文字列で対

応する変換指定子を持たないようにする方法があります。この呼び出し後のカウンタの値をチェックするだけで、実引数が評価されることを確認できます。以下がテストコードです。

```
#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;
}
```

以下は、この要件のテスト設計に対してテストを構築する方法の説明です。

```
/* テスト設計 REQ-C90:7.9.6.1-evaluate:
   書き込み用のファイルを作成し、fprintf()関数でそのファイルに文字列を出力する。この呼び出しには、最後に余分な実引数としてポストインクリメントされるカウンタを置き、その fprintf()関数の書式文字列に対応する変換がない場合でも、最後の実引数が評価されるか確認する。
*/
```

実引数の評価は、個別の fprintf()関数よりも言語の性質に関することであると主張することもできるでしょう。しかし、この要件は fprintf()に対して明示的に言及されているもので、そのことがこれを確認するよい理由となります。この要件を確認する二つ目の理由は経験に基づいたものです（上述の「エラー推測」を見てください）。printf()関数のファミリーはコンパイラによって最適化されることがよくあります。これらの関数の書式と実引数が上記のように単純化できる場合、コンパイラは呼び出しを単純な形式に置き換えることが多くあります。そのような printf()に由来する最適化では、この要件を満たす必要があります。

（printf は引数の個数が可変であり、評価の順序がどうなるかといった難しい問題があるので、この身近な関数のテスト例を介して SuperTest がよく考えて作られたものであることが示されます）

まとめ

C言語仕様は複雑であり、それを正しく実装することの重要性を考慮すると、標準ライブラリの実装が当然のものであるとは言えません。ライブラリが正しく実装されていることを確認する最善の方法は、規格から抽出された要件と、それら要件を満たすテスト設計の両方を含むテストスイートで認定することです。

FUJI SETSUBI
富士設備工業株式会社 電子機器事業部　www.fuji-setsu.co.jp