C/C++ code unit testing on a shoestring

Ark Khasin (akhasin@macroexpressions.com)

MacroExpressions

Abstract
Safety standards such as IEC 61508 make entry threshold to safety-related software (firmware) products rather high, including, not in the least, the effort required for unit testing. Automation tools do exist (e.g. LDRA Testbed or IPL Cantata++) but they cost arm and leg and have problems of their own, such as compiler adaptation and steep learning curve.

Much of the requirements can be met by abusing the C preprocessor of your very own standard-compliant C compiler. The technique grew out of the one I employed for my customer where it was sufficient to satisfy the requirements of SIL level 2.

This paper outlines some of the possible techniques in the hope that you may find them useful in evaluating your approach to unit testing and considering whether to go with a commercial test automation tool or to do it yourself.

The techniques focus on C code but are equally applicable to C++.

Unit testing requirements

There is more to unit testing of safety-related code than testing of input-state-output relationships: the test is supposed to look under the hood and demonstrate that the code execution path is as expected (and, presumably, as designed), that the results of important interim computations are correct and that all execution paths have been exercised.

The unit testing troika

The three horses that pull the cart of unit testing are:

- Test harness – a contrived code that executes the test cases which are invented to test the unit under test
- Test stubs – optional functions (or macros) created to replace functions (or macros) called from the UUT to abstract from the actual behaviors of the real functions.
- Test instrumentation – code plugged into the UUT to expose its behaviors normally not visible from the outside and to output "documentable" traces of execution

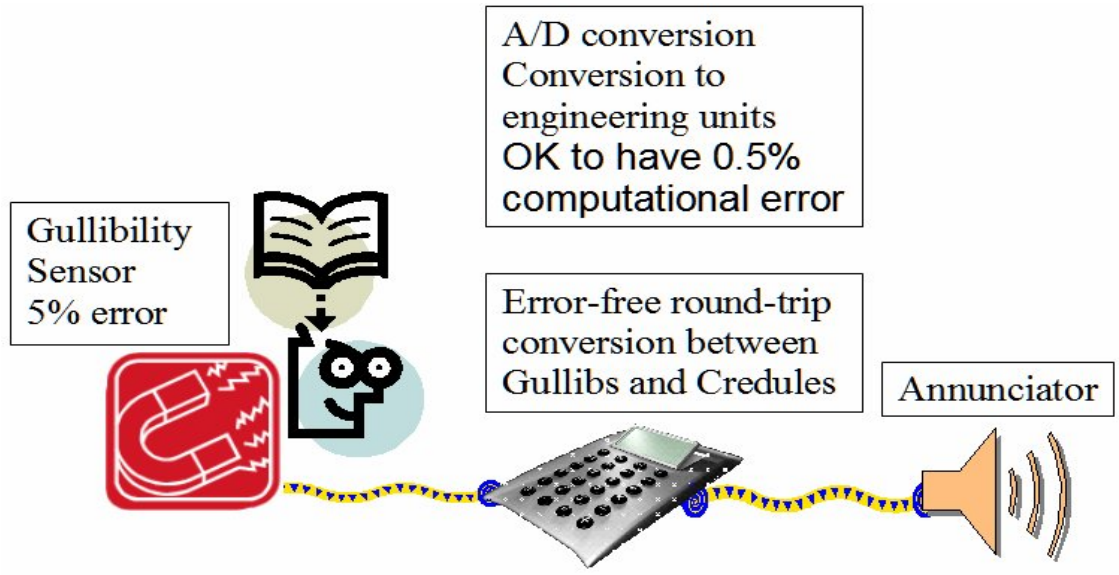
Harness

A decent test harness might consist of a common execution framework and a series of test cases pluggable into this framework but otherwise specific to the particulars of the UUT.

A test automation tool creates an execution framework for you. This is of course useful but the value of this service is not terribly high: You, all by yourself, can design and implement the framework once and be done with it.

Test cases are to be devised and coded according to the nature of the unit under test. This obvious truism allows, however, to put the claims of test automation tools in perspective. When a vendor says their tool will generate test cases for you, this may be so, but the cases generated in many (if not most) cases are not what you want. The reason is simple: the automation derives the test cases from analysis of your code and has no knowledge of the semantics implemented in it.

Take a simple example: you need to measure, say, hmmm… gullibility and raise an alarm if it exceeds a user-configurable threshold, entered in the units of either gullibs or credules.



Considering that the gullibility sensor and the A/D circuit have noise, you may decide, in the design phase, that converting raw A/D read to gullibs or credules may easily tolerate a fixed-point computation with an error of, say, five counts or e.g. 0.5%, as long as the computation itself is very fast and/or simple. On the other hand, you probably want the round-trip conversion of the threshold from gullibs to credules and back to gullibs to be error-free. (Otherwise, the user will unwittingly change the threshold by simply changing the units back and forth.)

The tests to cover this design are:

- Measurement conversion test. Verify that for all raw A/D values and other inputs (such as sensor calibrations) the result differs from a naïve double-precision calculation by at most five counts – or 0.5% as the case may be.

- Units conversion test. Verify that for all covered levels of gullibility the round-trip conversion of the engineering units yields the original value.

Chances are, your production code will have no traces of the design requirements (other than in comments, if you are particular enough). So it is unreasonable to expect test cases generated automatically for the tests identified above: you have to code these tests yourself. Note that it varies among the test automation tools how easy it is to integrate your own tests with your own acceptance criteria into the vendor's framework.

This is not of course to say that the no useful test cases can be generated automatically. For instance, generating tests for a state machine is quite possible – simply because all there is to a state machine ends up being in the code and can be analyzed.

Stubs

If a function you are testing calls a function in a different unit, you need to make a decision on the testing approach: You can create a stub to replace the called function with your own, or you can use the real function.

The decision depends on whether the UUT execution depends on what the called function does (think of, e.g., `strcpy`) or whether you merely delegate creating some side effects to that function. In the former case you obviously need a real (or an equivalent) implementation. In the latter case, a stub will do, but it must announce that it has been called.

Instrumentation

Creating instrumented code out of your production code comprises an intimidating amount of menial labor. It is here where the test automation tools ought to shine. They do, provided they parse your code and (important!) its dependencies correctly.

However, a lot of automation can be accomplished by using clever C macros. To get a taste of it, consider a definition in file scope (i.e., outside of any block)

```
static short foo;
```

The problem is to inspect its value, say, before and after executing a test case, without modifying your source file. While this seems impossible, we can do this if we have a macro like this:

```
#define static /*nothing*/
```

Note that it is a valid C macro which makes `foo` a variable with external linkage. Your harness code can now say:

```
extern short foo;
```

This trick won't work though for static objects in a block scope: the macro will make `foo` an automatic. This can be repaired by a variation on the theme:

```
#define static extern
```

This new macro turns an *uninitialized* definition into a declaration, and it turns out that it is legal in C to put declarations any place you could put a definition. The definition itself,

```
short foo;
```

will then go to your test harness file.
An *initialized* definition, like

```
static long bar = 42;
```

will remain a definition (albeit unusual) of the variable `bar`, so you will need a declaration in the harness file, like

```
extern long bar;
```

Note: This will break the compilation if you have a `static` variable defined with a block scope (i.e. within curly braces). That's because a definition of an external-linkage object is illegal with in a block.
This may actually be a good thing because there are arguments to be made against using `static` in block scope. Some coding standards prohibit this outright.

Other instrumentation techniques will be demonstrated in the next section. Our concern now will be, where to put the macro responsible for the trick
`#define static extern`

We do not want this definition visible anywhere except in the unit under test. To achieve this goal, let's do the following.

Assume that there is a header file in your project that is included in every source file. It is more than likely that you have one already; it might hold global project configuration parameters and/or include common goodies like `stddef.h` and `limits.h`. Let's say it is called `project.h`, so all sources have a statement

```
#include "project.h"
```

Let's now modify this ubiquitous `project.h` by adding the following:

```
#include "instrum.h"
```

This can be treated by project developers as a magic incantation; the header `instrum.h` shall have no effect on the normal build process. However, it is responsible for creating instrumentation when a source becomes the unit under test.

To achieve this variable behavior, we construct `instrum.h` to look as follows:

instrum.h

```
#ifdef INSTRUM_HEADER
#   include INSTRUM_HEADER

//abuse of static
#ifdef INSTRUM_STATIC
# define static INSTRUM_STATIC
#endif /*INSTRUM_STATIC*/

//abuse of other keywords
............

#endif /*INSTRUM_HEADER*/
```

(Here and elsewhere in this paper I omit the standard header guards to avoid clutter. They should be in place in real implementation.)

The idea of `instrum.h` is that in normal compilation of a source file, `INSTRUM_HEADER` is not defined and the source compiles as it always did.

When, however, a source file, say `foo.c`, is the UUT, we create an instrumentation header file for it, say, `instrum_foo.h` and pass the definition

```
INSTRUM_HEADER="instrum_foo.h"
```

on the command line of the compiler. (Usually, it's a –D compiler switch, or an equivalent configuration in the integrated development environment. Note that on Windows platforms passing a quoted definition in the command line is a tricky dealing with CMD.EXE; try `INSTRUM_HEADER="\"instrum_foo.h\""`.)

As we will see later, there is a good chance to use the same instrumentation header for all sources to be unit-tested, say, `instrum_uut.h`. This approach will be our target, and the definition to pass to the compiler is

```
INSTRUM_HEADER="instrum_uut.h"
```

The instrumentation header, `instrum_uut.h`, will look like so:

```
instrum_uut.h
-------------

#define INSTRUM_STATIC extern

//abuse of other keywords
............
```

(Or it may have
`#define INSTRUM_STATIC`
or no definition for `INSTRUM_STATIC` at all, depending on the needs.)

That is, if instrumentation for `static` is not defined, the keyword will keep its normal meaning. Otherwise, its meaning becomes whatever the instrumentation header assigns to it; this remains completely transparent to the magic header `instrum.h`.

C code instrumentation

Now we are in a position to devise instrumentation of the code by abusing other keywords.

Instrumenting the `if` statements

Following the pattern outlined in the previous section, let's add the following to `instrum.h`:

```
#ifdef INSTRUM_IF
#   define if(condition) INSTRUM_IF(condition)
#endif
```

The instrumentation header `instrum_uut.h` would have something like

```
#define INSTRUM_IF(condition) \
     if(instrum_if(#condition, (condition)!=0, \
     __FILE__, __LINE__, __FUNCTION__))

extern int instrum_if(const char *condition_name,
                      int condition,
                      const char *filename,
                      int line,
                      const char *function_name);
```

Implementation of the function `instrum_if` can be anything you want it to be, except that in order not to alter the behavior of you code, it must return its second argument (`condition`). For instance, the following implementation just prints what condition was evaluated and if it is true or false:

```
int instrum_if(const char *condition_name,
```

```
                int condition,
                const char *filename,
                int line,
                const char *function_name)
{
    printf("Condition %s in funtion %s"
        " (file %s line %d) is %s\n",
        condition_name, function_name,
        filename, line, (condition)?"true":"false");
    return condition;
}
```

This implementation goes to some instrumentation support source file. Of course this file itself should not be instrumented.

Some (pre-C99) compilers might not define `__FUNCTION__`, or you might not care to output the file name. Your implementation might be like so:

```
#define INSTRUM_IF(condition) \
     if(instrum_if1(#condition, (condition)!=0, \
     __LINE__))

extern int instrum_if1(const char *condition_name,
                       int condition,
                       int line);
```

with a corresponding implementation.

We will see later that instrumentation of other keywords will require a use of `if`. The `if` will come there instrumented but we don't want to see the effects of that instrumentation. So a real instrumentation function for `if` must detect whether the `if` is a result of instrumentation of something else and if so suppress all instrumentation actions.

Toward code coverage analysis

Assume for a moment that all your functions are written using only `if/else` statements to control program flow.

A test achieves 100% code coverage if and only if every `if` statement was executed and at least once the condition was false and at least once the condition was true (some people call this "branch coverage"). If you always strive for 100% coverage (as you should), the first condition can be dropped: A test achieves 100% code coverage if and only if the following holds: If an `if` statement was executed then at least once the condition was false and at least once the condition was true. This can be easily proved by induction on block nestedness level.

If your `INSTRUM_IF` macro's output uniquely identifies the specific `if` statement then it is pretty straightforward to analyze the overall test output to see if every `if` statement reported its condition both true and false at least once. Otherwise, a more involved instrumentation is necessary.

Instrumenting the `while` statements

The `while` keyword can be replaced in a similar manner. We put in `instrum.h`:

```
#ifdef INSTRUM_WHILE
#   define while(condition) INSTRUM_WHILE(condition)
#endif
```

And of course a mutatis mutandis entry is added to the instrumentation specific header such as `instrum_uut.h`.

```
extern int instrum_while(const char *condition_name,
                         int condition,
                         const char *filename,
                         int line,
                         const char *function_name);

#define INSTRUM_WHILE(condition) \
     while(instrum_while(#condition, (condition)!=0, \
     __FILE__, __LINE__, __FUNCTION__))
```

An implementation of the function `instrum_while` may, however, have a peculiarity.

There are two idioms, `while(1)` and `while(0)` which you may want to treat differently. The first one is a synonym for `for(;;)` which I don't particularly like but the fact is that some reputable people do use it. The second one is a part of the `do{...}while(0)` construct which is commonly used on two occasions: in macro definitions to wrap a block and straight in the code to make the `break` statements do the work of the dreaded `goto` <end-of-block>.

You might want to instrument `while(1)` as `for(;;)` and not instrument `while(0)` at all. Here is a simple implementation of not instrumenting these idioms:

```
int instrum_while(const char *condition_name,
                  int condition,
                  const char *filename,
                  int line,
                  const char *function_name)
{
    if(strcmp(condition_name, "0") != 0 &&
       strcmp(condition_name, "1") != 0) {
            printf("Loop cond. %s in function %s "
```

```
                    " (file %s line %d) is %s\n",
            condition_name, function_name,
            filename, line, (condition)?"true":"false");
    }
    return condition;
}
```

Of course an implementation like this should be kept in a file which is not instrumented.

Instrumenting the `switch` statements

If the goal of instrumentation is simply to announce the value of the controlling expression of a `switch` statement, we can follow the pattern established above and place

```
#ifdef INSTRUM_SWITCH
#   define switch(ctl_stmt) INSTRUM_SWITCH(ctl_stmt)
#endif
```

in `instrum.h` etc.

This would be good enough for establishing a regression base and using it in regression tests. However, if we want to prove code coverage, we need information on whether a given `case` or `default` was hit. That is to say, we need to instrument the `case` and `default` labels, which is our next subject.

Instrumenting the `default` statements

To instrument `default`, let's follow our usual pattern and put in `instrum.h`

```
#ifdef INSTRUM_DEFAULT
#   define default INSTRUM_DEFAULT
#endif
```

Now, to sensibly define `INSTRUM_DEFAULT`, we need to use the `default` keyword and to stick instrumentation code somewhere around. To do so, observe that if `falseval` evaluates to 0, then

```
default:
```

in any context is functionally equivalent to

```
default: if(!falseval) a_unique_label:
```

and to

```
default: if(falseval) ; else a_unique_label:
```

between which two I don't have a preference. (A unique label will be needed to consume a dangling colon. It will remain unused, and most compilers will issue warnings about unused labels. You can safely ignore or suppress them for the unit under test.)

For `falseval` we can take

```
(instrum_default(__FILE__,__LINE__,__FUNCTION__),
        !instrum_false)
```

where

```
extern void instrum_default(const char *filename,
                            int line,
                            const char *function_name);
```

is some function that e.g. announces hitting a `default` statement, and

```
extern int instrum_false;
```

is a variable with the value 0 in a different translation unit (so that the compiler is not tempted to optimize anything out).

Note that because of the comma operator, the whole expression for `falseval` must be parenthesized (because `if` is already a macro!)

For a unique label we can take a concatenation of the word `instrum_label` and the line number (e.g. a unique label for line 2007 will be `instrum_label2007`). This is a common C fare –

```
CAT(instrum_label, __LINE__)
```

where the `CAT` macro concatenates the two *expanded* arguments:

```
#define CAT1(a,b) a ## b
#define CAT(a,b) CAT1(a,b)
```

NOTE. You may choose to run your (instrumented) tests in the host environment when possible. If you use Microsoft Visual C++ to build the executable, the construction of the artificial label – `CAT(instrum_label_, __LINE__)` – may be broken because of the broken implementation of `__LINE__`. To fix this, you can remove the support for "Edit and Continue" (command-line option /ZI) in the project, or, for version 7.0 and above, use the non-standard `__COUNTER__` instead of `__LINE__`. Many thanks to Alf P. Steinbach for pointing it out.
(Of course, other compilers may have their idiosyncrasies, too.)

Now we are in a position to put the pieces together and to define `INSTRUM_DEFAULT` as follows:

```
#define INSTRUM_DEFAULT \
    default: \
    if((instrum_default(__FILE__,__LINE__,__FUNCTION__), \
        !instrum_false)) \
        CAT(instrum_label, __LINE__)
```

For this instrumentation to compile there cannot be two (or more) `default` labels on the same line (or else we'll produce two identical artificial labels). But for this to happen, there must be two or more `switch` statements on the same line, which is not a terribly good practice; we can ignore it. After all, if you do engage in this practice, a compilation error will notify you.

There is a useful variation of this instrumentation of `default`, especially for people who cannot stand the comma operator: Instead of making `instrum_default` a `void` function, make it a function returning an `int`, and implement it to always return a zero. It would then look like so:

```
#define INSTRUM_DEFAULT \
    default: \
    if(!instrum_default(__FILE__,__LINE__,__FUNCTION__) \
        CAT(instrum_label, __LINE__)
```

Instrumenting the `case` statements: a little help from a coding style needed

Our next step is to instrument the `case` labels. That is, we are going to create a macro `case`. For such a macro to be useful, we must somehow make the numeric label itself, however indirectly, to participate in the macro expansion. The only sensible way of achieving it that I found is to pass the label as a parameter to the macro. In other words, we want something like this:

```
#define case(x) case(x): SOMETHING(x)
```

With this macro, consider

```
    case MYCASE:
    case 2007:
    case (HERCASE):
    case (HISCASE):
```

The first two occurrences of are not recognized as macro calls (missing parameter list) and are not replaced. The third and the fourth occurrences are valid macro calls and will be expanded as desired.

Looking at this example from the vantage point of writing the code, we can conclude that for a `case` to end up being instrumented, its numeric label must be parenthesized. This, admittedly, is not a common practice. But that's how you need to write it to instrument the statements in order to prove code coverage. This is a matter of your team's coding standard.

Having resigned to instrumenting only `case`s with parenthesized labels, let's put a macro implementation together. Following the pattern, put in `instrum.h`

```
#ifdef INSTRUM_CASE
#   define case(x) INSTRUM_CASE(x)
#endif
```

For the implementation of `INSTRUM_CASE` in `instrum_uut.h`, observe that, just like for `default`, a passage

```
case(x):
```

is functionally equivalent to

```
case(x): if(!falseval) a_unique_label:
```

Following the same pattern as for `default`, we can do the following:

```
#define INSTRUM_CASE(x) \
     case(x): if((instrum_case(#x, x, \
     __FILE__, __LINE__, __FUNCTION__), !instrum_false)) \
        INSTRUM_CAT(instrum_label, __LINE__)
```

where `instrum_case` is an appropriately defined function. The type of the second parameter must be large enough to hold any label used in your application's switch statements. Hopefully, `long long` will do, if your compiler provides it.

As with `default`, it is easy to do away with the comma operator.

It is not uncommon to see several `case`s in a single line if they have common implementation, like

```
    case (HERCASE): case (HISCASE):
```

The instrumented code won't compile because of a duplicate definition of the artificial label. Again, if you want to use this instrumentation, make it a rule to place a `case` in its own line. If you don't, the compiler will notify you.

Instrumenting the `for` statement

The syntax of the `for` statement makes it difficult to invent an abusive instrumentation macro replacement in a generic way. It is not easy to get to the loop control expression which would be the key: recall that the purpose is to ensure that the control expression was at least once true and at least once false. About as much as one can do is this:

```
#ifdef INSTRUM_FOR
#define for(triplet) INSTRUM_FOR(triplet)
#endif
```

in `instrum.h`, and in `instrum_uut.h` put

```
#define INSTRUM_FOR(triplet) \
    for(triplet) \
        if((instrum_for(__FILE__,
        __LINE__, __FUNCTION__),instrum_false)) ; \
        else
```

Explanation of this macro goes exactly as that for `INSTRUM_DEFAULT` and is omitted here.

This instrumentation may be useful for regression testing. To make the instrumentation more useful for proof of code coverage, we need to resort to the help of the coding policy. We can require that instrumentable `for` statements have macro-ized controlling expressions, i.e., instead of writing

```
for(expr1; expr2; expr3)
```

we write

```
for(expr1; ISTRUE(expr2); expr3)
```

where the non-instrumented `ISTRUE` macro is defined as identity macro, i.e. `instrum.h` has

```
#ifndef ISTRUE
#define ISTRUE(e) e
#endif
```

An instrumented version in `instrum_uut.h` is e.g.

```
#define ISTRUE(e) \
  instrum_istrue(#e,(e),__FILE__,__LINE__,__FUNCTION__)
```

with an appropriate definition of the function `instrum_istrue`.

This affects the coding style even more intrusively than instrumentation of the `case` labels, and therefore may affect your decision on whether to use this instrumentation or not.

Instrumenting the break statements and others

There is nothing interesting to learn about a break statement other than that it was executed. A definition like this will do:

```
instrum.h
#ifdef INSTRUM_BREAK
#   define break INSTRUM_BREAK
#endif

instrum_uut.h
#define INSTRUM_BREAK \
    if((instrum_break(__FILE__,__LINE__,__FUNCTION__), \
    instrum_false)); else break
```

The purpose of the function `instrum_break` is merely to announce the execution of the corresponding `break` statement.

The explanation of this scheme exactly follows that of `default`, and is based on the observation that

```
break
```

in any context is functionally equivalent to

```
if(falseval) ; else break
```

It should be noted that instrumenting the `break` and `continue` statements adds nothing to code coverage analysis: each is the last executable statement in a conditional branch (or you have dead code which your compiler – or at least your Lint – will tell you about). So instrumenting the condition evaluation (and, for a `break` in a switch, the corresponding `case`) provides all the information needed. The same argument applies to `return` and `goto` statements.

That said, your coding policy may prohibit using `goto`, and maybe, even `continue`. In this case, you may want to instrument them to make the test case fail.

Still, all of `continue`, `goto`, `return` can be instrumented using the same scheme as `break`. Note though that instrumenting the `return` has a little peculiarity: copying the `break` instrumentation one to one will have the apparent effect in an instrumented function of an execution path without a return. This will (or at least could) elicit the compiler diagnostic. To correct this, use an endless loop in the false path, like

```
#define INSTRUM_RETURN \
    if((instrum_return(__FILE__,__LINE__,__FUNCTION__), \
    instrum_false)) {for(;;);} else return
```

Putting it all together

The framework

A reasonable framework can be based on the notion of a *test set* – a collection of tests covering one unit. A test in the set consists of:

- Description
- Acceptance criteria
- Test setup code (optional)
- A number of *test cases*
- Test cleanup code (optional)

A test case consists of:

- Description (optional)
- Parameters (optional)
- The number of repetitions
- Test case execution code (which actually exercises a function you're testing)

I shall not, of course, insult your intelligence by elaborating on how to model this framework in C and how to write the generic code executing a test set. A few pointers are due here though.

Instrumenting the unit under test

We have put together a magic whereby compiling the UUT with the definition

```
INSTRUM_HEADER="instrum_uut.h"
```

on the command automatically instruments the UUT. There can be legitimate cases, however, where the common instrumentation is not what you want (e.g., as we discussed in the beginning, `instrum_uut.h` has
`#define INSTRUM_STATIC /*nothing*/`
and you want
`#define INSTRUM_STATIC extern`

The solution is to invent your own instrumentation header `instrum_myown.h` and pass it as `INSTRUM_HEADER`. A preferred way is not to re-do all the work but to include `instrum_uut.h` in `instrum_myown.h`, undefine the inadequate implementation, and define it in an appropriate fashion, e.g.

```
#include "instrum_uut.h"

#undef INSTRUM_STATIC
#define INSTRUM_STATIC extern
```

Stubs calling the original function

Occasionally, you may want a stub for `foo()` to call the real implementation of `foo()`. A simple application of this is to announce that `foo()` was invoked.

This, too, can be done with `instrum_myown.h`, but this requires that `foo()` be defined outside the UUT.

Let's name the stub differently, e.g. `stub_foo()` and add the following to `instrum_myown.h`:

```
//fool the UUT by renaming
#define foo stub_foo

//(optional) prototype for foo()
// - turns into prototype for stub_foo
#include "foo.h"
```

This works provided the header containing the prototype for `foo()` is guarded

Now you can implement `stub_foo()` elsewhere as you normally would, and it may call the original `foo()`.

Producing the test set output

The purpose of the execution of a test set is to generate an output file. All output items should indicate whether it is produced by harness, instrumentation or a stub, for easier comprehension. Depending on the setup, the output should always produce either HTML output or plain-text output.

The (nicely formatted) HTML output can then be used for manual inspection of the execution results and for deciding whether the test set passed or failed, which is necessary if some acceptance criteria are manual. The HTML output can be easily equipped with additional information (date/time, user, unit under test, version etc.) and be presented to the auditor as part of test documentation.

The plain-text output can be used for regression testing (I optimized the code; does it still work as before?). It can also be used for post-processing of your choosing so that additional information can be extracted.

Acceptance criteria

Acceptance criteria should be stated for a test in advance; printing them (see next section) serves as documentation. They state when you consider a test case passed or failed, and they can be manual or automatic.

A manual criterion simply describes what is expected to come out of the test case; all such criteria are considered passed if you accept the test set output file as a reference. An example of a manual criterion is a notification that a certain function was called, or the lack of such notification.

An automatic criterion produces the expected output independently (like by a different algorithm of computations or as pre-tabulated values) and programmatically compares the result of the test case execution with the expected result. The pass/fail info should be printed with the test case output and propagate up to test and test set summary result.

Analyzing the output

Plain-text output is of particular interest for test coverage analysis.

Branch coverage

As discussed earlier, if your code consists only of `if/else` statements, 100% code coverage is achieved if controlling expressions in all `if` statements have been both true and false. Similarly, if your code doesn't use the `switch` statement, 100% code coverage is achieved if controlling expressions in all `if`, `while` and `for` statements have been both true and false, provided that there is no unreachable (dead) code. If there is, the compiler (or at least Lint) should inform you about that.

(Note however that if only 99.9% of controlling expressions have been both true and false, we cannot conclude that the code coverage is 99.9%: it can be less because of a variety of nested execution paths not covered at all.)

So long as each controlling statement is instrumented and is uniquely identified in the output, it is a matter of simple post-processing of the output file to prove (or disprove) that it was true and false at least once during test set execution.

Now let's add the `switch` statements to the mix. Assuming that all controlling expressions in all `if`, `while` and `for` statements had been true and false at least once, you achieved 100% code coverage if and only if each of the `case` and `default` statements had been hit at least once. (If you have `switch` statements without a `default`, it is considered not a good practice yet it can be dealt with. Still, this case is more complicated and is omitted here.)

In the output file, instrumented `case` and `default` statements that were executed would announce themselves. To verify that all of them were executed, you can scan the source of the UUT to extract the `case` and `default` statements and match them against their announcements in the test output; if each of them was announced, you've got 100%

code coverage, otherwise, you haven't. This can be done with a not-so-sophisticated script whose complexity may depend on whether or not you want to account for nested `switch` statements.

If you use `goto`, coverage analysis immediately falls apart. The remedy is of course not to use it (and, as described above, instrument `goto` to fail the test). The claim that "sometimes it is necessary" is no longer true, given today's compiler technology. The logic is,

- You know that `goto` is ugly but need to use it
- You know that ugly code should be extracted into a smallest encapsulating function
- This function can be written with `goto` replaced with a `return`.
- Let the compiler worry about optimization

Condition-type coverage

"Condition" in this context means an atomic Boolean expression in a compound Boolean "decision" (i.e. branch) expression in `if`, `while`, `do`/`while` and `for`. The following models of coverage are commonly used:

- Condition coverage: each condition has been true and false at least once
- Condition/Decision (CD) coverage – a union of condition and branch coverage
- Much touted Modified Condition/Decision (MC/DC) coverage – CD coverage where each condition is shown to affect decision independently
    - If the (atomic) conditions are not independent, you won't get MC/DC coverage.
    - But you can get away with Recursive MC/DC (below)

Recursive MC/DC

I'd like to introduce a model of coverage, Recursive MC/DC, which is a variation on the MC/DC theme but doesn't constrain the conditions so heavily.

- "Decision" is considered to be a chain of perhaps compound conditions linked with logical OR or logical AND.
- Conditions comprising the decision are treated as atomic for the purpose of MC/DC coverage of the Decision
- Any compound condition is (recursively) treated as a decision whose MC/DC coverage is sought while treating the constituent conditions as atomic

Support of Recursive MC/DC coverage analysis

With the help of coding standard, it is possible to instrument the code to produce sufficient output for Recursive MC/DC coverage analysis.

To do so, first let's use `and` and `or` instead of `&&` and `||` respectively, with the purpose of redefining of `and` and `or`. This is built into C++; for C, we need to include the standard header `iso646.h`.

Second, let's enclose operands of `and` and `or` in parentheses. E.g. instead of writing

```
a || b && c>5
```

let's write

```
(a) or ((b) and (c>5))
```

Some coding standards, such as based on MISRA, require about that much.

Now we can put in `instrum_uut.h`: something like that:

```
#ifdef __cplusplus
#define INSTRUM_AND(e) \
     and instrum_and(#e, (e)!=0, __LINE__)
#else
#define INSTRUM_AND(e) \
     && instrum_and(#e, (e)!=0, __LINE__)
#endif

extern instrum_and(const char *cond_name,
                   int condition,
                   int line);
```

As usual, `instrum_and` must announce and return the value of the second argument (`condition`). And, of course, we do a similar definitions for `or`.

Now, `in instrum.h`, we can put un-defining of what we intend to redefine:

```
#ifndef __cplusplus
#  include <iso646.h>
#  ifdef INSTRUM_AND
#    undef and
#  endif
#  ifdef INSTRUM_OR
#    undef or
#  endif
#endif
```

and put redefinitions in place:

```
#  ifdef INSTRUM_AND
#    define and(e) INSTRUM_END(e)
#  endif
```

and similarly for `or`.

Now, consider by the way of example, executing

```
if( (a) or (b) or (c))
```
where a, b and c are logical expressions.
In execution of instrumented code (where `if` and `or` are instrumented!),

- `(a) or (b) or (c)` is evaluated
- `if` is announced as true or false (Decision)

In the first step:
If `(a)` was true, `or(b)` is not evaluated and not announced
If `(a)` was false, `or(b)` is evaluated and announced as true or false, in which latter case `or(c)` is evaluated and announced.
You get coverage w.r.t. a, b and c if you have test cases where

- `or(b)` was not announced
- `or(b)` was announced true
- `or(b)` was announced false, and
  - `or(c)` was announced false
  - `or(c)` was announced true

Of course, "short-circuiting" behavior of logical OR allows the implementation of `or` to announce only true. Similar (dual) holds for `and`.

Recursive application of this logic to the constituents of the compound conditions demonstrates that we have enough instrumentation to analyze Recursive MC/DC coverage.

Limitations of the approach

The instrumentation techniques for code coverage analysis are not bullet-proof: they require that an announcement of a statement uniquely identifies it. A simple example of where it is not the case is a construct like
```
if(++x) a; else if(++x) ...
```
where it is not easy to come up with instrumentation of `if` which would ensure a unique identification of each `if` statement.

Normally, these cases can be addressed by the coding policy. E.g., MISRA wants a block to follow an `if` and a coding style usually wants a newline to precede or to follow an opening curly brace.

There are rare cases though where nested blocks with `if`s and loop and `switch` constructs comprise a macro definition, and occasionally such a macro has merits. When such a macro is expanded, all instrumentation functions will get the same `__FILE__`, `__LINE__`, and `__FUNCTION__` values, so unique identification may be tricky.

Secondly, there is no transparent way to instrument the ternary operator, which we conveniently ignored previously. For instance,
```
if(x) {y=u;} else {y=v;}
```
has the same meaning and result as

```
y=(x)?u:v;
```
The former case can be instrumented and analyzed whereas the latter cannot. A workaround lies in the coding policy: One can require using the ternary operator in all *non-constant* expressions with the `ISTRUE` macro (discussed with the `for` instrumentation), like so:
```
y=ISTRUE(x)?u:v;
```

Turning limitations into opportunities

All the limitations we have encountered, except the use of `goto`, whether fundamental (as with `catch`) or aesthetic yet error-prone (as with `case`), can be remedied fairly easily with automatic conversion of the unit under test into an instrumented source subject to actual test execution.
Such conversion requires a fairly basic parser of the language and remains independent of the peculiarities of your compiler and/or CPU.

This direction may be well worth pursuing provided there is sufficient interest.

Remarks on abnormal execution paths

So far, we've been discussing the normal control flow.

The C language allows only one exceptional control flow mechanism, namely, `setjmp`/`longjmp`. There is nothing special to be done to account for it since it appears as normal control flow based on the return value of `setjmp`.

The `try`/`throw`/`catch` exception mechanism of C++ is a much harder to deal with. We need to analyze whether each `catch` statement had been hit during the test set execution. It is tempting to do the same trick we used (effectively) for the `for` statement instrumentation:
```
#define catch(a) catch(a) if(instrum_catch()) {} else
```
Unfortunately, this won't even compile because `catch()` requires a compound statement to follow. At this point I don't know of a way of instrumenting `catch` by redefining the keyword.

Evaluating the commercial test automation tools

Now that you have an idea of what test automation you can get for free straight out of your compiler, the first question is, how much more functionality you get from the tool XYZ and whether it's worth the money – and the learning.

The next question is about usability of the tool XYZ, of course provided that it supports your compiler and your CPU. For instance, does it work smoothly with your version control system? (I know of at least one tool that doesn't like read-only files.) Is test report independent of the machine on which the test was executed? (Some tools annoyingly insist on absolute paths.) How easy is it to bring in your own test case in the framework generated by the tool? (Recall the gullibility example.)

Since the test execution usually takes some time, dependencies management in the tool are important. Does it know to rebuild and rerun the test if a source file of the unit changed? A header file on which the source depends? A stub?

Oh, and then of course there are bugs and your license reads "NO WARRANTY". The problem, and a conceptual problem at that, is that a bug may have catastrophic effect, e.g. if the tool says you have 100% coverage whereas you are nowhere near. (I have seen this with my own eyes.) Even if you discover a bad bug and report it to the vendor, you remain hostage of their release schedule.

If you are satisfied with the answers your prospective vendor has to offer, go for it. Otherwise, you may find the techniques outlined in this paper useful. To get started with the do-it-yourself approach, you can download Maestra – a free reference implementation from http://www.macroexpressions.com/dl/maestra.zip, or visit the home page http://www.macroexpressions.com/maestra.html.

A note on free unit testing frameworks

There are at least two applicable testing frameworks that are open-source and free of charge (CUnit and CppUnit; the latter can be adapted to testing C code). They deservedly gained a fair amount of acceptance; however they share some limitations:

- Dependence on dynamic memory management
- Need in target platform and compiler adaptation
- Tight integration of runtime test management and test result output
- Most importantly, lack of means of code coverage analysis

The first three attributes may pose problems in resource-constrained embedded environments. The last one is a problem in safety-related product development.

A conceptually simpler and more powerful way of organizing unit testing is

- Configuring test sets statically (at build time) to scale to the available computing resources
- Running a unit test is to produce an output file (essentially, an execution log) that lends itself to post-processing on a host platform.

Such decoupling is what (some of) those expensive tools aim to do. It may be important if e.g. test output is sent via serial interface and captured on a host machine. For instance, the Maestra reference implementation uses `printf` to output readable text, but it doesn't have to be so: you can output tokens of any sort (even binary) to reduce the raw output size, and then post-process it into a readable form.

About the author

Ark Khasin, PhD, is with MacroExpressions (http://www.macroexpressions.com). He can be contacted at akhasin@macroexpressions.com.