YAMI4

Test coverage report

For YAMI4Industry, v.1.3.1

Table of Contents

Document scope

The purpose of this document is to provide the evidence of test coverage that is achieved with the help of the test suite that was prepared for the YAMI4Industry package.

This report focuses on statement coverage and MC/DC coverage for conditional statements.

The document applies to the 1.3.1 version of the YAMI4Industry package.
See the following web site for general information on the project:

http://www.inspirel.com/yami4/industry.html

Coverage reporting method

The level of source code coverage of the test suite is computed with the help of several testing methods:

- automatic instrumentation of source code for branch naming, branch tracing and decision coverage
- regular unit testing
- unit testing with focus on conditional coverage and MC/DC coverage
- fault injection to simulate failing system calls

Automatic instrumentation of source code relies on a script that parses the original source code and generates two instrumented versions:

- source with marked named branches – this version is distributed to users as part of regular library package where branch names can be used for reference or for traceability,
- source with print statements for tracing execution of each named branch – this version is not distributed within the regular package, but is available on demand together with the complete test suite that exercises the library code and that causes print statements to leave trace in the program output, which allows to associate the execution of the test with the library source structure.

The source code instrumentation can be explained with the help of code example:

```
/* in source file.c */
void foo(int i)
{
    /* some statements */
```

```
    /* ...             */
    if (i > 0)
    {
        /* some conditional statements */
        /* ...                         */
    }
    else
    {
        /* alternative */
        /* ...         */
    }
}
```

When the original source file `file.c` is processed by the code instrumentation script, the following two versions are created – the marked source code:

```
/* marked file.c */
void foo(int i)
{
    /* some statements */
    /* ...             */
    if (i > 0)
    {
        /* branch foo_c1 */
        /* more conditional statements */
        /* ...                         */
    }
    else
    {
        /* branch foo_c2 */
        /* alternative */
        /* ...         */
    }
}
```

and the instrumented version, in a separate directory:

```
/* trace instrumented file.c */
void foo(int i)
{
    puts("file.c:foo");
    /* some statements */
    /* ...             */
    if (i > 0)
    {
        puts("file.c:foo_c1");
        /* more conditional statements */
        /* ...                         */
    }
    else
    {
        puts("file.c:foo_c2");
        /* alternative */
        /* ...         */
    }
}
```

The first version, with named branches, contains additional comments that do not interfere with normal program execution and as such can be considered to be identical to the original source code. This version is included in the source distribution package.

The second version, with tracing statements, prints messages with file name and branch name every time the execution passes through some branch. Since the code is written in C, there are no implicit execution paths (like exceptions or compiler-generated function calls) and therefore all branches are guaranteed to leave their trace when executed and at the same time the printed trace is a 1:1 evidence of the test coverage.

In addition to the above two code versions, the instrumenting script prepares an index file containing all branch names.

Branches are named by appending consecutive numbers to the name of enclosing function, with branch nesting represented by chaining numbered segments.

It is instructive to note that the top-level branch of any given function does not need to be marked with additional comment, but every nested branch is explicitly named. The instrumented version contains print statements at both top-level branches (where the branch name is just a function name) and within each nested branch.

Conditional branches (that is, those that are bodies of `if` or `switch` statements) have letter `'c'` as a segment prefix. Thus, `"foo"` is the first (top-level) branch of function `foo`,

while `"foo_c2"` is the second conditional branch of the `"foo"` branch - it is also convenient to state that `"foo"` is a parent of both conditional branches `"foo_c1"` and `"foo_c2"`.

Compound conditions are manually annotated in original source files with the use of comments that are recognized by the script, which can then generate more complex tracing instructions that print the name of the condition, the outcome of the whole expression as well as its parts. These traces can be later used to make claims related to the MC/DC coverage.

When unit tests are executed, they leave in their output stream the exact trace of all executed branches. It is then possible to analyze the trace in order to find out how many times the branches were exercised by the tests.

Based on this information, every known branch (from the index file) gets the following mark:

- `'-'` if the non-conditional branch was never executed or if the conditional branch has a parent that was never executed,
- `'T'` if the branch was executed as many times as its parent branch (in the case of conditional branches, like `"foo_c1"`, this means that the condition was always true),
- `'F'` if the conditional branch was never executed, but its parent branch was executed at least once (that is, the condition was always false when it was checked)
- `'C'` if the conditional branch was executed at least once, but less often than its parent branch (this means that the condition was tested for both true and false outcomes).

For non-conditional branches it is desired to obtain the `'T'` mark, which simply indicates that the given code was executed by the test driver.

For conditional branches it is desired to obtain the `'C'` mark as an evidence for condition coverage (that is, to show that the conditional statement was tested for both true and false outcomes).

As an example, if the function foo above is tested with positive value of its actual parameter, the trace left by the test will be:

```
file.c:foo
file.c:foo_c1
```

Note that trace `"file.c:foo_c2"` will not appear in such test.

Similarly, if foo is tested with some negative value or 0, the trace will be:

```
file.c:foo
file.c:foo_c2
```

And then trace `"file.c:foo_c1"` will not appear in the output.

Both tests have to be executed and the traces have to be accumulated to achieve the desired branch marks:

- `'T'` for `file.c:foo`
- `'C'` for `file.c:foo_c1`
- `'C'` for `file.c:foo_c2`

Such marks allow to claim 100% code and condition coverage testing.

The accumulating of branch traces is automated by another dedicated script, which allows to produce total marks for all known branches.

Regular unit tests focus on successful code paths that represent typical and intended usage scenarios. If these are not sufficient for 100% branch coverage, additional tests are written that allow all conditional branches to execute - these are typically related to the handling of corrupted input data or invalid API call sequences, which provides the evidence that the library is robust and immune to improper use.

In order to cover also the hypothetical failures of system calls, the testing approach relies on simple fault injection technique where wrapper functions are used instead of direct system calls.

For example, instead of the direct call to the `close` system function:

```
cc = close(file_descriptor);
```

the call to wrapper function is used:

```
cc = test_close(file_descriptor);
```

where `test_close` has the same signature as the target system function and can therefore be used as a drop-in replacement of the original system call.

There are as many wrapper functions as there are system calls used in the library code.

These wrapper functions use shared variables that are set up by unit tests to decide when the actual system call should be used or when the failure should be simulated instead. This way the unit test drivers can exercise all error-handling branches of the library code without further instrumentation of the underlying operating and run-time systems.

The traces for compound conditions are produced together with branch traces (they are interleaved in the output stream) and are analyzed separately by means of manual inspection, which is justified by the fact that the number of compound conditions in the library code is very small.

Test coverage results by module

The following sections list the results obtained for each module (source file, listed in alphabetic order) comprising the YAMI4Industry package. For each module the total number of branches, number of non-conditional and conditional branches are given with the explanation of the MC/DC analysis, if applicable.

File agent.c

This module implements the logic of the YAMI4 agent object, which encapsulates the resource management and handles outgoing and incoming message routing.

The `agent-test.c` file is a dedicated test driver for this module.

There are 84 branches in this module.

There are 30 non-conditional branches, all with mark `'T'`.

There are 54 conditional branches, all with mark `'C'`.

There is one compound condition in this module, in function `yami_do_some_work`:

```
/* decision channel_fault_when_doing_work:
                (res == yami_io_error) ||
                (res == yami_channel_closed) ||
                (res == yami_not_enough_space)                          */
/*                                    A: res == yami_io_error          */
/*                                    B: res == yami_channel_closed    */
/*                                    C: res == yami_not_enough_space */
/* end                                                                 */
if ((res == yami_io_error) ||
    (res == yami_channel_closed) ||
    (res == yami_not_enough_space))
{
    /* branch c2_1_c3 */
```

The test log contains the following related distinct traces:

```
decision channel_fault_when_doing_work: F, A: F, B: F, C: F
decision channel_fault_when_doing_work: T, A: F, B: F, C: T
decision channel_fault_when_doing_work: T, A: F, B: T, C: F
```

```
decision channel_fault_when_doing_work: T, A: T, B: F, C: F
```

The above output demonstrates complete MC/DC test coverage of this compound condition.

File channel.c

This module implements the frame handling of a single connection. Channels are responsible for buffer management and partial message transmission. There can be many active channel objects in a single agent and they can operate independently as well as in full-duplex mode.

The `channel-test.c` file is a dedicated test driver for this module.

There are 59 branches in this module.

There are 20 non-conditional branches, all with mark `'T'`.

There are 39 conditional branches, all with mark `'C'`.

There are no compound conditions in this module.

File network_utils.c

This module encapsulates network-related services and allows to hide the details of the POSIX interface (that is, all references to POSIX API are hidden in this module).

The `network_utils-test.c` file is a dedicated test driver for this module.

There are 175 branches in this module.

There are 23 non-conditional branches, all with mark `'T'`.

There are 152 conditional branches, all with mark `'C'`.

There is one compound condition in this module, in function `yami_wait_for_work`:

```
/* decision has_place_to_check_listener:
        (has_some_free_channels != 0) &&
            (listening_sock != INVALID_FILE_DESCRIPTOR)             */
/*                      A: has_some_free_channels != 0             */
/*                      B: listening_sock != INVALID_FILE_DESCRIPTOR */
```

```
/* end                                                              */
if ((has_some_free_channels != 0) &&
    (listening_sock != INVALID_FILE_DESCRIPTOR))
{
    /* branch c2 */
```

The test log contains the following related distinct traces:

```
decision has_place_to_check_listener: F, A: F, B: T
decision has_place_to_check_listener: F, A: T, B: F
decision has_place_to_check_listener: T, A: T, B: T
```

The above output demonstrates complete MC/DC test coverage of this compound condition.

File options.c

This module implement a single initialization function that provides default values for the agent configuration options object.

The `options-test.c` file is a test driver for this module.

There is only 1 branch in this module.

It is a non-conditional branch with mark `'T'`.

There are no compound conditions in this module.

File serialization.c

This module implements all routines that are responsible for serializing and deserializing data in memory buffers, in line with the YAMI4 wire protocol. Both elementary (like serialization of single data items) and high-level (like serialization of complete headers for different message types) functions are implemented in this module.

The `serialization-test.c` file is a test driver for this module.

There are 223 branches in this module.

There are 51 non-conditional branches, all with mark `'T'`.

There are 172 conditional branches, all with mark `'C'`.

There is one compound condition in this module, in function yami_parse_frame_header:

```
/* decision frame_header_is_valid: (frame_id == -1) &&
        (message_id_tmp >= 0) && (message_header_size_tmp > 0) &&
            (frame_payload_size_tmp > 0)                          */
/*                              A: frame_id == -1                */
/*                              B: message_id_tmp >= 0           */
/*                              C: message_header_size_tmp > 0   */
/*                              D: frame_payload_size_tmp > 0    */
/* end                                                           */
if ((frame_id == -1) && (message_id_tmp >= 0) &&
    (message_header_size_tmp > 0) && (frame_payload_size_tmp > 0))
{
    /* branch c2_c1 */
```

The test log contains the following related distinct traces:

```
decision frame_header_is_valid: F, A: F, B: T, C: T, D: T
decision frame_header_is_valid: F, A: T, B: F, C: T, D: T
decision frame_header_is_valid: F, A: T, B: T, C: F, D: T
decision frame_header_is_valid: F, A: T, B: T, C: T, D: F
decision frame_header_is_valid: T, A: T, B: T, C: T, D: T
```

The above output demonstrates complete MC/DC test coverage of this compound condition.

File utils.c

This module implement several helper routines that allow the whole library to be completely independent from the C run-time library.

The `utils-test.c` file is a test driver for this module.

There are 49 branches in this module.

There are 22 non-conditional branches, all with mark `'T'`.

There are 27 conditional branches, all with mark `'C'`.

There are two compound conditions in this module:

1. in function yami_strcmp:

```
/* decision strcmp_finished: (c1 != c2) || (c1 == (int32_t)'\0') */
/*                       A: c1 != c2                             */
/*                       B: c1 == (int32_t)'\0'                  */
/* end                                                           */
if ((c1 != c2) || (c1 == (int32_t)'\0'))
{
    /* branch 1_c1 */
```

   The test log contains the following related distinct traces:

```
decision strcmp_finished: F, A: F, B: F
decision strcmp_finished: T, A: F, B: T
decision strcmp_finished: T, A: T, B: F
decision strcmp_finished: T, A: T, B: T
```

   The above output demonstrates complete MC/DC test coverage of this compound condition.

2. in function `yami_string_to_uint32`:

```
/* decision uint32_parse_invalid_char:
                  (c < (int32_t)'0') || (c > (int32_t)'9') */
/*                                  A: c < (int32_t)'0'     */
/*                                  B: c > (int32_t)'9'     */
/* end                                                      */
if ((c < (int32_t)'0') || (c > (int32_t)'9'))
{
    /* branch 1_c1 */
```

   The test log contains the following related distinct traces:

```
decision uint32_parse_invalid_char: F, A: F, B: F
decision uint32_parse_invalid_char: T, A: F, B: T
decision uint32_parse_invalid_char: T, A: T, B: F
```

The above output demonstrates complete MC/DC test coverage of this compound condition.

Summary

The results presented in previous sections allow to claim 100% test coverage for code (branches) as well as MC/DC for compound conditions.

Revision history

Revision	Comment
1	Initial revision, refers to the 1.3.0 version of the YAMI4Industry source package and its related test suite.
2	Updated for the 1.3.1 library version.