

Testing

STATE-DEPENDENT CODE

Objectives

At the end of this section, you will be able to

- Create a test for a state machine, where tests results depend on the state machine's internal state

testIDEA

Contents

STATE-DEPENDENT CODE

1	Concept of a state machine	3
2	Example for the state machine	4-6
3	Initialization of the state machine	7-8
4	Testing get-function	9
5	Testing state changes	10
6	Create final state	11
7	Overview test vectors	12
8	What about coverage?	13
9	Summary	14-15

testIDEA

1 CONCEPT OF A STATE MACHINE

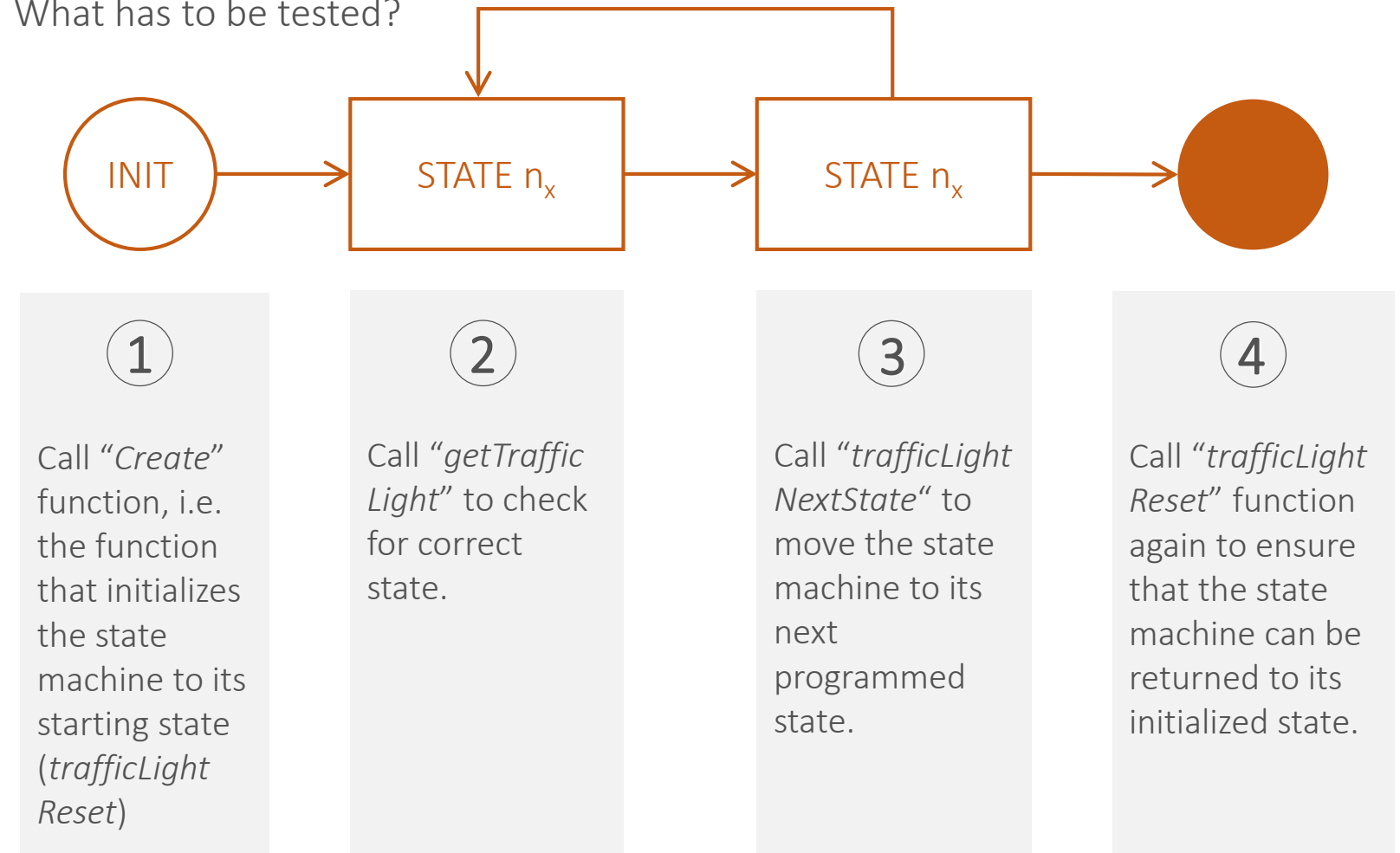


A state machine is, by its very nature, an object whose next state change is dependent upon its current state. Any unit testing platform for such programming constructs must ensure that this state information is maintained between tests.

One recommend test strategy here is a test framework using the CRUD (Create, Read, Update, Release) approach. This allows for testing code that can have states and outcomes that vary for the same function calls.

There are parallels here to the testing of C++ that we saw in Unit 07. C++ had its class object that was maintained across method calls during testing; a state machine will, likewise, need its state data maintained across function calls.

What has to be tested?



2 EXAMPLE FOR THE STATE MACHINE

The example code for this unit represents the state changes of a traffic light.

The different traffic light states are defined in an *enum*. As mentioned previously, this is preferable coding strategy to using *#define* and testIDEA is additionally able to access these symbols for use in the tests.

```
enum TrafficLightState {
    STATUS_INIT,
    STATUS_RED,
    STATUS_RED_AMBER,
    STATUS_GREEN,
    STATUS_AMBER,
    STATUS_ERROR_1,
    STATUS_ERROR_2
};

static TrafficLightState _trafficLightState =
STATUS_INIT;
static TrafficLightState _trafficLightNextState =
STATUS_INIT;
static TrafficLight _trafficLight = LIGHT_ALL_OFF;

void trafficLightReset(void) {
    _trafficLightState = STATUS_INIT;
    _trafficLightNextState = STATUS_INIT;
}
```

2 EXAMPLE FOR THE STATE MACHINE

The State Machine has the following transitions:

Init → Amber

Amber → Red

Red → Red_Amber

Red_Amber → Green

Green → Amber

Additionally, two error states are defined providing a flashing amber light in the event that an error should occur.

```
void __attribute__((noinline))
trafficLightNextState(void) {
    _trafficLightState =
        _trafficLightNextState;

    switch (_trafficLightState) {
        case STATUS_INIT:
            _trafficLightNextState =
                STATUS_AMBER;
            break;

        case STATUS_RED:
            _trafficLightNextState =
                STATUS_RED_AMBER;
            break;

        case STATUS_RED_AMBER:
            _trafficLightNextState =
                STATUS_GREEN;
            break;

        case STATUS_GREEN:
            _trafficLightNextState =
                STATUS_AMBER;
            break;

        ...

        case STATUS_AMBER:
            _trafficLightNextState =
                STATUS_RED;
            break;

        case STATUS_ERROR_1:
            _trafficLightNextState =
                STATUS_ERROR_2;
            break;

        case STATUS_ERROR_2:
            _trafficLightNextState =
                STATUS_ERROR_1;
            break;

        default:
            _trafficLightState =
                STATUS_ERROR_1;
            _trafficLightNextState =
                STATUS_ERROR_2;
    }
}
```

2 EXAMPLE FOR THE STATE MACHINE



Testing state machines with testIDEA is a little bit tricky. Depending on the size of the state machine and the project, a model-based testing approach might be more appropriate.

Our traffic light project has a modest complexity and thus a very limited number of test cases. Thus we can demonstrate the testing of this state machine quite simply just with testIDEA.

```
TrafficLight __attribute__((noinline))
getTrafficLight(void) {
    TrafficLight returnValue =
    LIGHT_ALL_OFF;

    switch (_trafficLightState) {
        case STATUS_INIT:
            /* Do nothing - return
            default lights
            off state */
            break;

        case STATUS_RED:
            returnValue =
            LIGHT_RED;
            break;

        case STATUS_RED_AMBER:
            returnValue =
            LIGHT_RED_AMBER;
            break;

        case STATUS_GREEN:
            returnValue =
            LIGHT_GREEN;
            break;

        case STATUS_AMBER:
            if (_trafficLightState == STATUS_ERROR_1) {
                returnValue = LIGHT_AMBER;
            } else if
            (_trafficLightState
            == STATUS_ERROR_2) {
                returnValue = LIGHT_ALL_OFF;
            } else {
                returnValue = LIGHT_AMBER; }
            break;

        default:
            _trafficLightState = STATUS_ERROR_1;
            _trafficLightNextState = STATUS_ERROR_2;}

    return returnValue;}


```

3 INITIALIZATION OF THE STATE MACHINE



First of all we have to create the first part of the testing framework:

We create an executed test case on base test level for the initial state. In our example this initial state is created by the function *trafficLightReset*.

Create **first executed test on base test level** to initialize state machine
→ *trafficLightReset*

New test case wizard

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:

Expected result

☐ Default expression for function return value test

☒ Custom expression and function return value name

Expression:

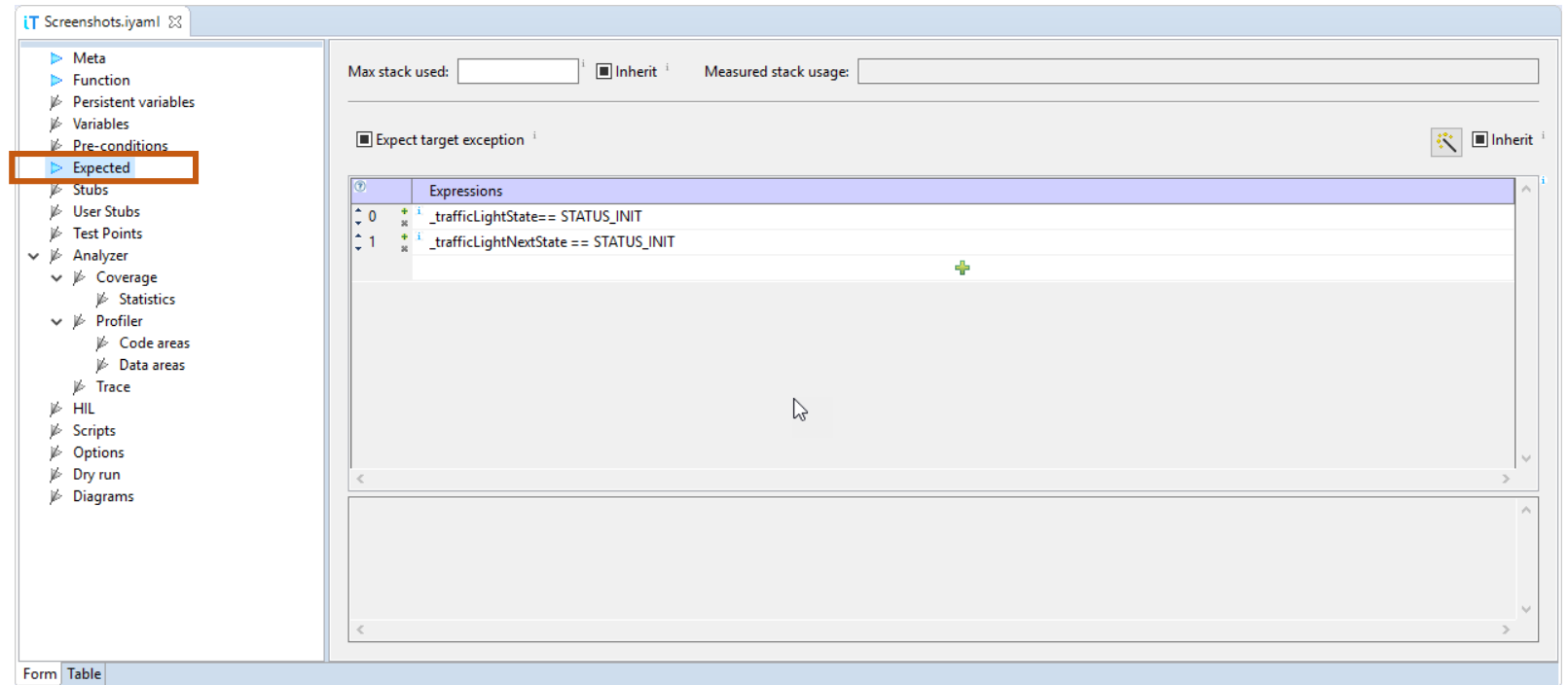
Ret. val. name:

< Back Next > **Finish** Cancel

3 INITIALIZATION OF THE STATE MACHINE



In this first test case we expect that the traffic light is initialized in the correct manner, e.g. that *STATUS_INIT* is stored in the internal variables of the state machine.



	Expressions
0	<code>_trafficLightState == STATUS_INIT</code>
1	<code>_trafficLightNextState == STATUS_INIT</code>

4 TESTING GET-FUNCTION



The next step is to create a second executed test vector at base test level which is used to test the *getTrafficLight* function.

To test this function we have to enter the function name as well as the expected value (*STATUS_INIT*).

Create **second executed test on base test level** to ensure that get-function returns expected values
→ *getTrafficLight*

New test case wizard

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:

Expected result

☒ Default expression for function return value test

☐ Custom expression and function return value name

Expression:

Ret. val. name:

Enter expected function return value. This value will be used to automatically generate expression '`_sys_rv == <value>`' in section 'Expected'. For example, if you enter:
10
expression '`_sys_rv == 10`' will be automatically generated. This feature can only be used for scalar types (char, int, ...). For complex types specify Ret. val. name and expression below. Additional expressions can later be entered in section 'Variables'.

5 TESTING STATE CHANGES



Next we create a base test for the *trafficLightNextState* function, which will not be executed, and the accompanying derived tests with the changing expected states.



By deriving these tests from a base test, it makes it easier to modify or replace this section should the state machine functionality ever change in the future.

Create **third (non executed) base test and derived tests** to check internal local variable status after each call
→ *trafficLightNextState*

id	func			testTimeout	coreId	assert	
	func	params +	retVal			isExpectException	
							0
0	<i>trafficLightNextState</i>						
1	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_INIT
2	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_AMBER
3	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_RED
4	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_RED_AMBER
5	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_GREEN
6	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_AMBER
7	<i>trafficLightNextState</i>						<i>_trafficLightState</i> == STATUS_RED

6 CREATE FINAL STATE



We will then create one more executed test at base test level to set the state machine into a final state.

In this example the final state is moved into by a call to *trafficLightReset*. This leaves the state machine in its reset state.

Create **fourth executed test on base test level** to ensure that the state machine can be returned to its initial state
→ *trafficLightReset*

7 OVERVIEW TEST VECTORS



After all the test cases have been created, the structure should look as shown opposite.

The screenshot displays the testIDEA interface. On the left is a project tree with the following structure:

- vyj0uvsw0vc0 : trafficLightReset
- vyj0x2jod0g0 : getTrafficLight
- vyj0zefatcjk : trafficLightNextState
 - vy16pa9qq3sw : /
 - vy16pa9r0gio : /
 - vy16pa9rar4w : /
 - vy16pa9re6ow : /
 - vy16pa9rh9s : /
 - vy16pa9rh9u : /
 - vy16pa9rh9w : /
- vyj0uvsw0vc0 : trafficLightReset

In the center is a settings panel with the following options:

- ☐ Meta
- ☒ Function
- ☐ Persistent variables
- ☐ Variables
- ☐ Pre-conditions
- ☒ Expected
- ☐ Stubs
- ☐ User Stubs
- ☐ Test Points
- ☐ Analyzer
 - ☐ Coverage
 - ☐ Statistics
- ☒ Profiler
 - ☐ Code areas
 - ☐ Data areas
- ☐ Trace

On the right is a test vector table with the following data:

	func	assert
	func	expressions
0	trafficLightReset	$_trafficLightState == STATUS_INIT$ $_trafficLightNextState == STATUS_INIT$
1	getTrafficLight	$_sys_rv == STATUS_INIT$
2	trafficLightNextState	
3	trafficLightNextState	$_trafficLightState == STATUS_INIT$ $_trafficLightNextState == STATUS_AMBER$
4	trafficLightNextState	$_trafficLightState == STATUS_AMBER$
5	trafficLightNextState	$_trafficLightState == STATUS_RED$
6	trafficLightNextState	$_trafficLightState == STATUS_RED_AMBER$
7	trafficLightNextState	$_trafficLightState == STATUS_GREEN$
8	trafficLightNextState	$_trafficLightState == STATUS_AMBER$
9	trafficLightNextState	$_trafficLightState == STATUS_RED$
10	trafficLightReset	$_trafficLightState == STATUS_INIT$ $_trafficLightNextState == STATUS_INIT$

8 WHAT ABOUT TEST COVERAGE?

These tests and this strategy most likely provides close to a 100% code coverage. However, the coverage of the functionality is not fully guaranteed. This is because we haven't checked that we can reset the state machine successfully at any point in its operation, or that more than one cycle of the state machine can be achieved. This highlights the challenges faced by those developing Unit Tests.

iSYSTEM would recommend evaluating a Model Based Testing approach to test case generation to ensure that such software is fully tested. Such an approach is possible using the software tools from [sepp.med GmbH](#), such as [MBTsuite](#).

Close to a 100% code coverage,

but:

what about test coverage?

(reset possible at any point, more than one cycle, ...)

—————> Model Based Testing approach recommended
for more complex state machines

SUMMARY

testIDEA

- The testing process for a low-complexity state machine contains four steps:
 - Testing the initialization of the state machine
 - Testing any “getState” functions
 - Testing the “state change” functions
 - Testing any other state machine features (error handling, etc.)
- It is recommended to use a model based testing approach for more complex state machines

