OCS Test Guidelines

Release 1.6.2

Software and Controls Group

Dec 10, 2019

Contents

CHAPTER

ONE

INTRODUCTION

Each module of the OCS, like any other subsystem of the GMT is required to provide a method for the verification of it's requirements. The most common methods of verification are demonstration, test, inspection and analysis. Although the OCS modules include both hardware and software components, these guidelines only apply to the verification of requirements that involve software components. Following is a brief description of the verification methods [SEBoK] :

- Inspection: Technique based on visual or dimensional examination of an element. Inspection is generally non-destructive, and typically includes the use of sight, hearing or simple physical manipulation. No stimuli (tests) are necessary. In the case of the OCS software modules this affects mostly the visual aspects of the user interface, for example, confirming that the requested user interface elements are present. Usually, no specific tests have to be created as the execution of software applications in the presence of the users may be enough. In the context of the OCS, this method often involves the inspection of the module as a whole.
- Analysis: Technique based on analytical evidence obtained without any intervention on the element under test using mathematical or probabilistic calculation, logical reasoning, modeling and/or simulation. Mainly used where testing to realistic conditions cannot be achieved or is not cost-effective. How to implement the analysis method of an OCS software module and how to preserve its embodiment is beyond the scope of these guidelines and most likely should be coordinated with Systems Engineering department.
- Demonstration: Technique used to demonstrate correct operation of the element under test against operational and observable characteristics whitout using physical measurements (no or minimal instrumentation or test equipment). Demonstration is sometimes called 'field testing'. It generally consists of a set of tests selected by the user to show that the element response to stimuli is suitable or to show that operators can perform their assigned tasks when using the element. Observations are made and compared with predetermined/expected responses. As in the case of of the inspection method, no expecific tests have to constructed or executed. In the context of the OCS, this method often involves the demonstration of the module as a whole.
- Test: Technique performed onto the element under test by which functional, measurable characteristics, operability, supportability, or performance capability is quantitatively verified when subjected to controlled conditions that are real or simulated. Testing often uses special test equipment or instrumentation to obtain accurate quantitative data to be analyzed. This verification method is specially relevant to the test of software elements in the domain of control systems. It's also relevant as it is the main verification method used to verify the Components and Packages that form a OCS Module. The tests that we discuss in these guidelines belong mostly to this category.

Note: Note that in this context, requirements is used as a general term, as in software systems requirements are also expresed as features or user stories.

CHAPTER

TWO

TYPES OF TESTS

As mentioned above the test method involves the verification of functional, operational or performace capabilities of the element under test. For the purpose of the verification of the OCS system and its modules we defined three categories of tests:

- Functional tests: These tests are necessary to verifiy the functional capabilities of the element under test. These capabilities are often captured as functional or operational behaviors in formal system engineering or user stories in agile software engineering. The tests must fulfill the following guidelines:
  - Should be executed automatically when possible
  - Shall have positive and negative cases
  - Shall be able to establish if the element behaves according to specification
  - The source and version of the functional test specification shall be recorded
- Performance tests: These tests are necessary to verify the performance capabilities of the element under test. The tests must fulfill the following guidelines:
  - Should be executed automatically when possible (it may depend on the measurement method)
  - The method to measure performance shall be characterized
  - The uncertainty in the measurement shall be specified for each performance tests
  - The source and version of the performance test specification shall be recorded
- Interface tests: These tests are necessary to verify the ability of the element under test to be integrated with other elements. The interface tests must fulfill the following guidelines:
  - Should be executed automatically when possible
  - The interface test shall be able to establish if the interface is present
  - The interface test shall be able to establish if the interface conforms to the protocol specification
  - The interface test doesn't check the correctness of the behavior of the element under test
  - The source and version of the interface test specification shall be recorded

CHAPTER

THREE

TYPES OF ELEMENTS UNDER TEST

The OCS modules are built up from a hierarchical composition of *Components* and *Packages*. The test strategy is thus bottom-up, where elementary components are tested (analogous to unit testing) prior to being integrated in aggregated entities. The following guidelines shall be fulfilled by each type of element:

- Component testing
  - There shall be an independent test for every Component Feature
  - A Component is considered tested when all its Feature tests are performed
- Package testing
  - There shall be at least one integration test per Package
  - A Package is considered tested when all its Component tests are performed and the Package integration tests are performed
- Subsystem testing
  - There shall be at least one integration test per Subsystem
  - A Subsystem is considered tested when all its Package tests are performed and the Subsystem integrated tests are performed

---

Note: Subsystems defined in the GMT model are mapped into Modules for the purpose of development, testing and integration. As explained in the Software and Controls Standards, Subsystems are modeling entities and are decomposed in Packages and Components.

Note that in this context, Component Feature is used as defined in the OCS metamodel: an elementary capability of a Component, e.g. a property, a state variable, an alarm event

---

CHAPTER

FOUR

GENERAL GUIDELINES

- Every test (Feature, Component, Package, Subsystem) must include positive and negative tests
- The results of the tests must be reported in a standardized manner by creating a TestRecord data structure:

attribute	description
name	Name of the test
desc	Description of the test
req_id	Id of the requirement verified by the test
target_type	Type of the element under test
test_type	Type of the test
input	Input data to the test when applicable (e.g. parameter file)
output	Output data to the test when applicable
url	URL of the element under test
expected	Expected value
result	Result of the test
timestamps	Sequence of timestamps. The first indicates the start of the test, the last indicates the end of the test. The intermediate timestamps can be used to inject additional time probes during the execution of the test (e.g. comunication send and receive events in a distributed test).

```
test_record:
    name:          "Feature a (positive)"
    desc:          ""
    req_id:        ""
    target_type:   /feature|component|package|subsystem/
    test_type:     /interface|correctness|performance/
    input:         {}
    output:        {}
    result:        /PASS|FAIL/
    expected:      {}
    url:           ""
    timestamps:    []
```

- Test records shall be saved as files during the test run or stored a test database
- The following table defines which tests are considered mandatory

target_type / test_type	interface	functional	performance
feature	–	yes	–
component	–	yes	yes*
package	yes**	yes	yes*
subsystem	yes	yes	yes*
system	–	yes	yes*

* only when performance requirements are specified ** only when packages are procured by different entities

CHAPTER

FIVE

IMPLEMENTATION

The OCS development environment provides a set of tools to support the definition, creation and execution of tests.

- The *gds gen* command generates tests skeletons based on the model definition of a module.
- The OCS Test Framework implements a set of classes that facilitate the execution of distributed tests.
- The *gds test* command allows the automatic execution of test.

5.1 Test Generation

The *gds gen* command is able to generate test skeletons based on the model definition of any model element. If the model element is an aggregate (e.g. Package, Subsystem) it will generate also the tests of the elements that are part of that model element.

The generated skeletons are created in the test directory of the module:

```
$GMT_LOCAL/modules/<module>/
   |-- model/
   |-- src/
   |-- test/
```

Note: In the version 1.5 of the SDK, complete skeletons are only generated for elements of the class *Component*. For elements that are aggregated the skeleton includes an example of how to invoke an external test.

The generated tests are organized in the file system according to the following structure:

```
$GMT_LOCAL/modules/<module>/test/coffee/
   |-- <module>_<pkg_1>_pkg/
   |     |-- <module>_<cmp_1>_cmp/
   |     |       |-- <subs>_<cmp1>_functional_01_test.coffee
   |     |       |-- <subs>_<cmp1>_performance_01_test.coffee
   |     |-- ...
   |     |-- <module>_<cmp_n>_cmp/
   |     |
   |     |-- <module>_<pkg_1>_interface_01_test.coffee
   |     |-- <module>_<pkg_1>_functional_01_test.coffee
   |     |-- <module>_<pkg_1>_performance_01_test.coffee
   |-- ...
   |-- <module>_<pkg_n>_pkg/
   |-- <module>_interface_01_test.coffee
   |-- <module>_functional_01_test.coffee
   |-- <module>_performance_01_test.coffee
```

- The following example generates the skeletons for the DCS subsystem *hdk_dcs*

```
gds gen --target test hdk_dcs
```

Note: The gds command will generate a warning when the model includes an element that is not supported for test generation yet, such as the Application element in the hdk example. This is expected behavior.

The previous command will generate several test skeletons for the hdk_dcs module:

```
$GMT_LOCAL/modules/ocs_hdk_dcs/
   |-- test/
   |      |-- coffee/
   |      |      |-- hdk_ctrl_pkg/
   |      |      |      |-- hdk_main_ctrl
   |      |      |      |      |-- hdk_main_ctrl_01_functional_test.coffee
```

5.2 Test Framework

The Test Framework provides several classes that facilitate the creation and execution of distributed test

As can be seen in the example below, the skeleton includes the following:

- The skeleton class *HdkMainCtrlTestClient* that extends the superclass *TestClient*. The skeleton redefines two methods:

  start: For 'one shot' tests, write the test block in this method. If it is feasible here is where the element(s) under test can be spawned. If the element(s) under test are executed in a different computer, they could be started executing a remote shell command, which can be invoked with the node method spawnSync.

  step: For iterative tests, write the test block in this method

- An instance of a command line application *CoreCLIApplication* that provides access to the console and to command line options
- An instance of a *CoreContainer* which will direct to the console or to a file any core service event generated by the test client (e.g. logs, alarms, telemetry)
- An instance of the test client class *HdkMainCtrlTestClient* that is initialized with the conjugated of the element under test ports. Notice that the url's generated are those defined in the model class of the element under test, as the test generator doesn't have knowledge of which instance to test, as a consequence these urls need to be redefined with the ones corresponding to the instance that we intent to test. A more refined strategy would be implemented in future releases.

  Edit the file *ocs_hdk_dcs/test/coffee/hdk_ctrl_pkg/hdk_main_ctrl/hdk_main_ctrl_01_functional_test.coffee* to contain urls that correspond to the ones defined in *ocs_hdk_dcs/src/etc/conf/hdk_ctrl_pkg/hdk_main_ctrl/hdk_main_ctrl_config.co*
  For example:

```
# Component hdk_main_ctrl functional test

{ CoreContainer
CoreCLIApplication } = require 'ocs_core_fwk'
{ TestClient }       = require 'ocs_test_fwk'
```

```
class HdkMainCtrlTestClient extends TestClient

   start: ->
      super()
      @log.info @name, "Running functional test 01"      # Substitute␣
↪number with actual one
      # Include here the test section

   step: ->
      super()
      # Include here any periodic test


app         = new CoreCLIApplication  null, { name: 'hdk_main_ctrl_test_
↪client_app'} #, logging:  'metric'}
cntr        = new CoreContainer app,  null, { name: 'hdk_main_ctrl_test_
↪container' }

# Note 1: The test client ports are conjugated of the component under␣
↪test ports
# Note 2: This configuration shall be updated in case a specific instance␣
↪configuration of the Component under test is used
# Note 3: To execute this test run: > gds test --type functional hdk_main_
↪ctrl --number 01

test_client = new HdkMainCtrlTestClient cntr,
   properties:
      uri:                 {name: "uri",         default_value: 'gmt://
↪127.0.0.1:19000/test_fwk/test_test_pkg/hdk_main_ctrl_test_client'}
      name:                {name: "name",        default_value: 'hdk_main_
↪ctrl_test_client'}
      host:                {name: "host",        default_value: '127.0.0.1
↪'}
      port:                {name: "port",        default_value: '19000'}
      scan_period:         {name: "scan_period", default_value: 1000}
      acl:                 {name: "acl",         default_value: 'PRIVATE'}
   inputs:
      hmi_outputs_req:     { name: 'hmi_outputs_req',       port_type:
↪'pull',  url: 'tcp://127.0.0.1:8104', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      motor_ctrl_req:      { name: 'motor_ctrl_req',        port_type:
↪'pull',  url: 'tcp://127.0.0.1:8105', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      sdo_config_req:      { name: 'sdo_config_req',        port_type:
↪'pull',  url: 'tcp://127.0.0.1:8106', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      heartbeat_out:       { name: 'heartbeat_out',         port_type:
↪'pull',  url: 'tcp://127.0.0.1:8108', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
      hmi_value:           { name: 'hmi_value',            port_type:
↪'pull',  url: 'tcp://127.0.0.1:8122', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      motor_value:         { name: 'motor_value',          port_type:
↪'pull',  url: 'tcp://127.0.0.1:8123', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      temperatures_value:  { name: 'temperatures_value',   port_type:
↪'pull',  url: 'tcp://127.0.0.1:8124', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
```

```
      ops_state_value:      { name: 'ops_state_value',        port_type:
↪'pull',  url: 'tcp://127.0.0.1:8119', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
      sim_mode_value:       { name: 'sim_mode_value',         port_type:
↪'pull',  url: 'tcp://127.0.0.1:8120', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
      control_mode_value:   { name: 'control_mode_value',     port_type:
↪'pull',  url: 'tcp://127.0.0.1:8121', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
   outputs:
      hmi_inputs_val:       { name: 'hmi_inputs_val',         port_type:
↪'push',  url: 'tcp://127.0.0.1:8101', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      motor_state_val:      { name: 'motor_state_val',        port_type:
↪'push',  url: 'tcp://127.0.0.1:8103', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      temperatures_val:     { name: 'temperatures_val',       port_type:
↪'push',  url: 'tcp://127.0.0.1:8102', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      hmi_goal:             { name: 'hmi_goal',               port_type:
↪'push',  url: 'tcp://127.0.0.1:8116', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      motor_goal:           { name: 'motor_goal',             port_type:
↪'push',  url: 'tcp://127.0.0.1:8117', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      temperatures_goal:    { name: 'temperatures_goal',      port_type:
↪'push',  url: 'tcp://127.0.0.1:8118', blocking_mode: 'async', max_rate:␣
↪1000,  nom_rate: 1     }
      ops_state_goal:       { name: 'ops_state_goal',         port_type:
↪'push',  url: 'tcp://127.0.0.1:8113', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
      sim_mode_goal:        { name: 'sim_mode_goal',          port_type:
↪'push',  url: 'tcp://127.0.0.1:8114', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }
      control_mode_goal:    { name: 'control_mode_goal',      port_type:
↪'push',  url: 'tcp://127.0.0.1:8115', blocking_mode: 'async', max_rate:␣
↪1,     nom_rate: 1     }

app.setup()
app.start()
```

Once the test has been modified it can be processed and installed by executing the command *webpack* in $GMT_LOCAL/ocs_hdk_dcs/test/coffee

5.3 Test Execution

In order to execute the previous test we write:

```
> gds test hdk_main_ctrl --type functional --number 01
```