Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Test automation module in the INTO-CPS Platform

Deliverable Number: D5.2a

Version: 1.1

Date: 2016

Public Document

http://into-cps.au.dk

Contributors:

Adrian Pop, LIU
Florian Lapschies, VSI Oliver Möller, VSI

Editors:

Oliver Möller, VSI

Reviewers:

Julien Ouy, CLE
Thierry Lecomte, CLE
Etienne Brosse, ST
Martin Peter Christiansen, AI

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	2016-06-15	Adrian Pop	Initial document version
0.2	2016-07-29	Oliver Möller	Drafted outline and responsibilities
1.0	2016-10-31	Oliver Möller	Completed for Review
1.1	2016-11-29	Oliver Möller	Work in Review Comments

Abstract

This deliverable documents the implementation of a test automation module, which allows definition and configuration of automated test suites based on co-simulation model descriptions.

The implementation aspects focus on the test automation part of the evolving INTO-CPS application. This will grow throughout Years 2 and 3 and be extended to include more desired functionalities as the case studies progress.

The general test automation concepts and usage are explained. This includes identification and configuration of the system under test component, which can be a (model-based) simulation, or a manual or automated implementation (software in the loop or hardware in the loop). We discuss the incremental growth of a test project, that is extended with more goals and aspects until the desired test coverage is reached.

As a guiding example through these stages, a simple water tank controller is used.

The dependencies of artifacts in this process are governed by OSLC, which is highlighted in the appropriate places.

Contents

List of Figures

1 Introduction

This deliverable documents the implementation of a test automation module, which allows definition and configuration of automated test suites based on co-simulation model descriptions.

Starting from a modelling tool, we discuss how to translate this into a test project that is subsequently extended by more and more behavioural explorations, grouped into test procedures. Each test procedure covers a subset of possible behaviours, which is then accumulated into a list of test results that evaluate expected versus observed behaviour.

Each of the following tools will be integrated in the test automation module:

- Modelio `http://www.modelio.org/`
- RT-Tester [Ver15a, Ver15b], `https://www.verified.de/products/`

1.1 Running Example: Water Tank Controller.

As a running example we use the *water tank controller* that is also described in related INTO-CPS documents. We use a simplified version that only features one tank, see Figure 1.

The *Tank* has the capacity for a certain amount of water. Water may flow in at an unspecified rate. Water may flow out, if the *Valve* at the lower part of the tank is opened. A *Controller* has the goal[1] to keep the water level between a constant *Lower Bound* and a constant *Upper Bound*. It has a *sensor* input that measures the current water level and an *actor* output, which corresponds to closing or opening the Valve.

We specify the behaviour of the Controller such that it shall close the Valve if the water level drops to (or below) the Lower Bound. If the Upper Bound is reached (or exceeded), the Valve shall be opened.

For simplicity we work with the values **1** for the Lower Bound and **2** for the Upper Bound. Water level 0 corresponds to an empty tank.

[1] This goal is only achievable under certain side conditions, like limited water input etc. We explain this in the assumptions.

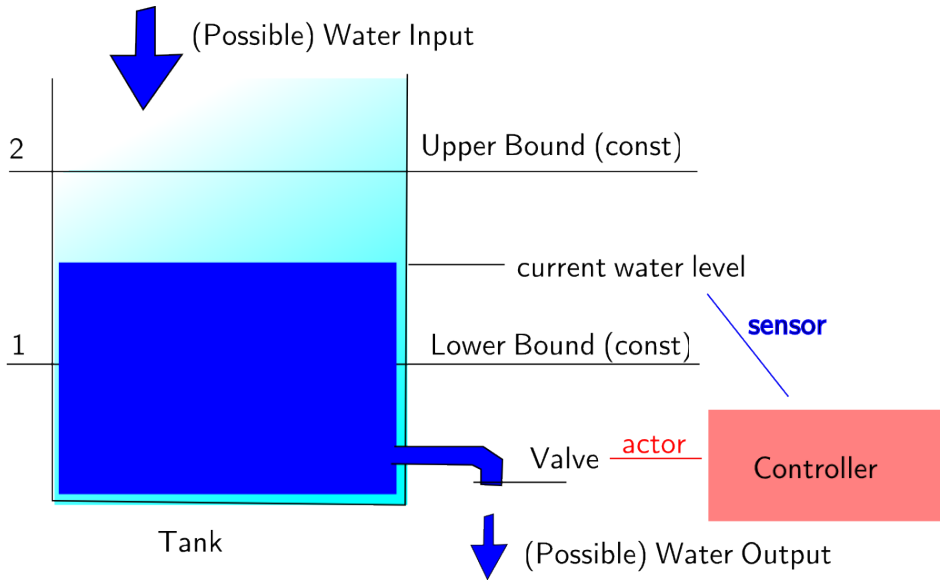

Figure 1: Water Tank and Controller.

Assumptions. We disregard situations where the Tank falls dry or the "Water Input" flow exceeds the maximal "Water Output" capacity over time: in these situations it is obvious that the goal "current water level between the bounds" cannot be achieved by the Controller.

We are only interested in the behaviour of the Controller for "well behaved" systems and how this can be meaningfully tested.

Water Tank Test Model. A possible *test model* for the Controller is depicted in Figure 2.

Independent from an implementation (which might be a simple threshold switch), the control logic is represented by a timed state machine.

In our test model, the "`Waiting`" state is idle until the timer t elapses. Every 1000 milliseconds, the state ("`Responding`") is entered, which evaluates the water level `wt3_level`. As can be seen in the transitions, there are three ways to go back to "`Waiting`":

- If the water level is above 2, the valve is opened, i.e., `wt3_valve` is set to 0.
- If the water level is between 1 and 2, no action is taken.

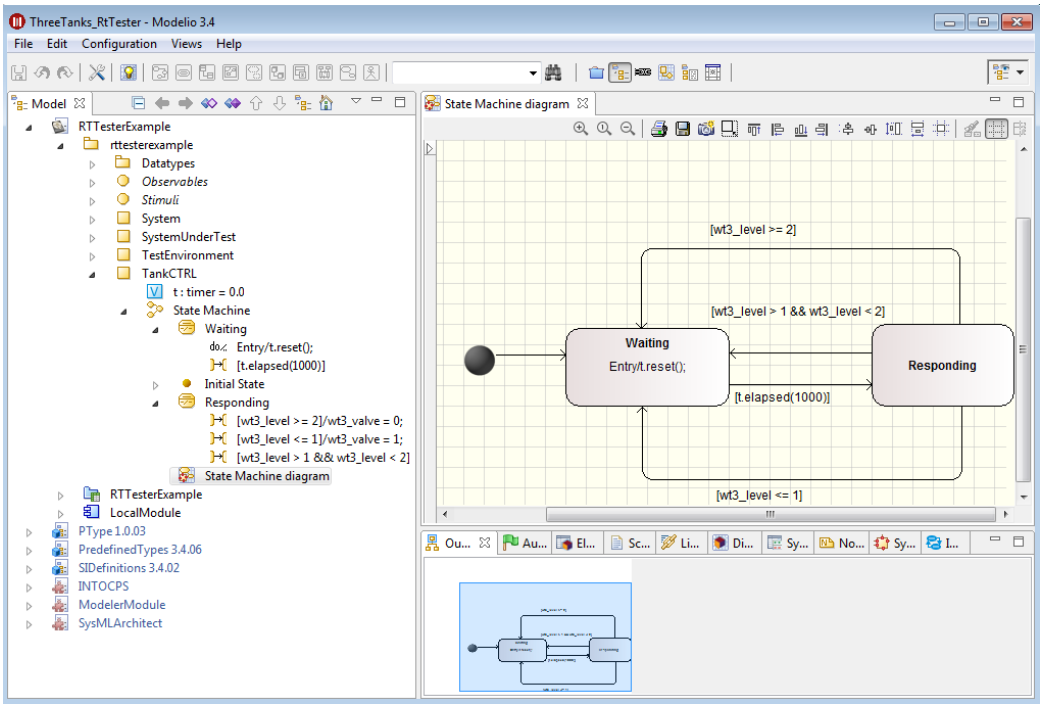

Figure 2: Water Tank Controller Test Model as State Machine (in Modelio).

- If the water level is below 1, the (drain) valve is closed, i.e., `wt3_valve` is set to 1.

Note that the guard conditions are both *mutually exclusive* and *complete*, which means that *exactly one* of the guards will be enabled. This guarantees deterministic progress of the model, no matter what water level is currently measured.

1.2 Test classifications: HiL, SiL, and MiL.

Test activities are often classified according to the nature of the test configuration. In particular the terms *Hardware-in-the-Loop*, *Software-in-the-Loop*, and *Model-in-the-Loop* - or HiL/SiL/MiL for short - are widely used for this.

The distinction of HiL/SiL/MiL is not always clear, because there are situations where more than one of the terms seems to be appropriate. For our purposes, it suffices to separate these terms as follows.

HiL (Hardware in the Loop)

There is (target) hardware involved, thus the FMU is mainly a wrapper that interacts (timed) with this hardware; it is perceivable that realisation heavily depends on hardware interfaces and timing properties.

SiL (Software in the Loop)
The object of your test execution is an FMU that is filled with some sort of software implementation of (parts of) the system. It can be compiled and run on the same machine that the COE runs on and has no (defined) interaction other than the FMU-interface. It does not matter (much) where this implementation comes from.

MiL (Model in the Loop)
The test object of the test execution is a (design) model, represented by one or more FMUs.[2] This seems similar to the SiL (if e.g., the SUT is generated from the design model), but *can* also imply that running the SUT-FMU has a representation on model level. For example, a playback functionality in the modelling tool could be used to visualise a test run.

This is to be understood as a *working definition*, we are aware that it will not completely fit.

1.3 Requirements

The following requirement are quoted from the description of work D7.3 [**?**].

Requirement 0024 A distributed and simulation network for Test Automation (TA) must be developed. Especially in the case of real-time HIL testing, where some parts of the co-simulated environment are replaced by real components, the test environment must be able to perform all simulation steps in real-time. While the test model still only specifies the intended system behaviours, the real-time co-simulated environment, which will be distributed on multiple hosts (e.g. real-hardware, testing host, co-simulation host), will have additional latencies introduced by the COE and by interfacing with the real hardware. As a result the test environment must be able to get information about the latencies from the COE and deal with them while evaluating test results.

[2] This is the case if in the water-tank model the test are run against "Simulation" instead of "SUT"; here, the "Simulation" is derived directly from the test model, which defines the behaviour of the FMU.

Requirement 0025 RT-Tester must provide an INTO-CPS FMI tool wrapper that is compliant with the COE. This allows for monitoring and stimulating outputs and inputs of other co-simulated models from the COE. An integration of the FMI interface allows RT-Tester to access available data using a standardised interface. This gives the test engineer the possibility to select any inputs/outputs of the co-simulated models and perform tests using data from those interfaces.

Requirement 0026 Integrating RT-Tester into the COE differs from integrating other co-simulations is one respect: RT-Tester will not introduce any new signals on the FMI interface, but only read outputs or stimulate inputs from generated simulations (e.g. 20-sim, OpenModelica). Therefore RT-Tester should be able to derive interface descriptions from the Modelio configuration of the COE and automatically derive a corresponding FMI interface from that. The RT-Tester test model will then be able to refer to all those identified interfaces, when the intended system behaviour of the co-simulation environment is specified, which will be used for test data generation.

Requirement 0027 When executing a whole test suite for a co-simulation environment, the goal is to leave all components completely unchanged for a test run. Only the generated test data and by this the simulated behaviour will change, as the test environment acts as the test driver to steer the simulation into a desired system state, which will be used to check all aspects of the modelled system. Therefore the test enviornment (TE) needs a way of stimulating inputs/outputs of other simulations, but also to control the COE, e.g. be restarting a co-simulation or restarting a specific model simulation. Use cases for this are for example power-on tests of models, where different power-up scenarios are tested.

Requirement 0028 RT-Tester must be able to monitor the complete execution of multiple co-simulated models and in parallel evaluate the monitored trace against the designed test model. The monitoring should happen without actively influencing the test execution, but just passively monitor the observations and perform checks on the monitored traces.

1.4 Related Work

A substantial amount of literature is available for model-based testing and cyber-physical systems, for specific pointers see, e.g., [**?**] and [**?**]. Since this work addresses several key issues (modelling, distribution, tracing), it would not be helpful to start citing long lists of literature here. Rather, it makes sense to relate this work to recent research projects with similar aim and scope.

In this respect, the following projects can be considered to be closely related.

ADVANCE[3] addresses analysis of connected CPSs, but focuses on formal verification rather than on testing.

AMADEOS[4] is also concerned with the time aspects of system of systems, which relates to the timed nature of the tool chain described here.

MODRIO[5] addresses the requirement modelling aspect, but not to the extend that it is confirmed by tests.

DESTECS[6] provided definitions for many concepts re-used by INTO-CPS.

COMPASS[7] and **PTOLEMY**[8] both address the distributed aspects with mixture of models of computation, without casting this into the FMI 2 co-simulation standard.

Further (less closely) related work can be found in [**?**, Appendix B].

2 Test Automation Capabilities

This section gives a brief introduction to test automation goals and limits. It is meant to clarify concepts and position the test automation activity in the INTO-CPS workflow. This is elaborated further in the subsequent sections.

For test automation, the following two aspects have to be identified.

[3] http://www.advance-ict.eu/
[4] http://amadeos.imag.fr
[5] https://itea3.org/project/modrio.html
[6] http://www.destecs.org/
[7] http://www.compass-research.eu/
[8] http://ptolemy.berkeley.edu/publications/index.htm

1. A *specification* of the desired behaviours, which is typically annotated with system requirements. In order to serve as a *test model*, it has to be defined as a timed state-chart.
2. An *implementation* of these behaviours, which may exist entirely in software (SiL), as a hardware deployment (HiL), or itself as a model (MiL), compare Section 1.2.

2.1 Considerations for Lifecycle Management

The central part of the test automation is the definition of the *test model*[9] All test automation activities depend on it and every change necessarily invalidates earlier test results.

The typical workflow for test automation is as follows.

1. Define (or modify) test model
2. Import that model in a test project
3. Configure a test procedure
4. Run a test procedure against an Implementation— the *System Under Test (SUT)*—and evaluate the result
5. Either
   (a) Continue with configuration of the next test procedure, or
   (b) Correct the SUT and retry, or
   (c) Refine (or correct) the test model

For this it is necessary to keep track of versions (like the version of the test model) and activities. With respect to Open Services for Lifecycle Collaboration (OSLC)[10], the following actions have been identified to be traced.

Define Test Model This includes all modelling activities that concern the test model, including definition of the state machine, association of system requirements with model elements, or changes in the graphical representation.

The end-point of this activity is the hand-over of the test model to a test project (via XMI export → XMI import).

[9] A specific example for a test model is given in Section 3.2.
[10] see `http://open-services.net/`

Define Test Objectives For each test procedure in a test project, test objectives need to be defined. This is done by configuring the test procedure such that it covers a certain set of test cases. Also the assumptions (on input values) and parameters (like acceptable latency of output values) need to be specified, see Section 5.2 for more details.

The end-point of this activity is the generation of a (runnable) RT-Tester 6 test procedure, which is encapsulated in a FMU.

Run Test This activity includes the configuration of the test run, including the FMU that acts as SUT (e.g., a wrapper to a SUT prototype implementation or a Simulation derived from the test model), the duration of the run, and the step size. The actual run is then performed by means of the COE.

The end-point of this activity is the actual termination of the test run.

For the actions above prototypical support has been implemented, using INTO-CPS OSLC messageFormatVersion 0.1.[11]

2.2 Current Limitations

The following capabilities apply to the current version of the tool chain. They are expected to be removed in the further progress of the project

1. Test configurations only support one SUT and one Test Driver/Simulator.

   Reason. Test data generation needs to identify the border between items that can be manipulated (environment) and items that cannot (SUT, possibly also simulations).

   Conceptually this means that if more than one SUT is selected, a new border has to be computed that puts all of the SUTs to one side of the equation. For this the tool support is currently missing.

2. No HiL example has yet been tried.

   Reason. HiL requires specific hardware setup; this has to be provided by the case studies (WP1).

3. System Requirement Tracing not available yet.

[11] This is ongoing work and subject to improvements.

Reason. The Modelio 3.4.1 export does not contain the requirement information; also, tracing support for RTT Windows has to be added.

3 Identifying Model Elements for Test Automation

Starting point for test automation are functional model elements that exhibit (timed or un-timed) behaviours that are subject to operational verification.

The *specification* part of test automation is necessarily done with aid of a modelling tool.

Purpose of a Test Model. A recurring question is why we have to construct a test model. If we already have (often painfully) constructed a system (design) model that captures the behaviours, why no use this as test model?

The purpose of a test model is to define (in a compact way) the behaviours that can be expected from a system part and subject this to automated inspection. If the design model (which is used to generated an implementation) is used for this, there a successful test confirms that the process for generating the implementation worked as expected. If the implementation is derived automatically, you are testing the code generator.

Deriving a test model from (parts of) the system requirements has the advantage that you can ignore or simplify aspects of the design that are owed to architectural decisions, e.g., deployment in several functional groups. Only the input/output of a test model has to match the one of the designated SUT. If it eases the testing process, several test models (one for each aspect) may be constructed for the same system.

Therefore it is an *option* but not an *obligation* to keep the test model separate from the design.

3.1 Relating Requirements to Model Parts

Currently, requirements are modelled at the higher level via the Modelio tool. The Modelio tool also supports linking of SysML requirements to SysML

blocks directly and these requirements can be exported via the XML Metadata Interchange (XMI) files to other tools such as RT-Tester. RT-Tester can read in these requirements and allow the user to bind test-cases to them.

Work is ongoing to globally relate requirements to model parts, test cases, simulation results and other artifacts via the traceability daemon, see D4.2d INTO-CPS Traceability Design.

The traceability daemon uses OSLC specifications. For requirement traceability the OSLC Requirements Management (RM) specification is relevant [12].

Some of these OSLC relations we plan to explore for test automation are:

1. Modelio: Requirement is linked to SysML block in ASD
2. RT-Tester: Define test mode, Define test objectives, Run test

Traceability in terms of test-case and requirement management is a core feature of RT-Tester tool chain. Connecting test cases and requirements, the associated status accounting (including reporting) during a test campaign are described in detail in [Ver15a] and [Ver15b].

In the future, the Modelio and RT-Tester will export the defined requirements and their links to model elements and test cases to the traceability daemon so that the requirements can be traced globally.

3.2 The Test Model - SysML Perspective

We use the test model of the water tank controller described on page 9, see Figure 2.

Why is this not a design model? Let us reflect on some properties of the state machine layed out above.

1. It uses an (internal) timer `t`, see guard "`t.elapsed()`"
2. The reaction to a raise or fall in the water level does not happen immediately; only ever seconds, the value of `wt3_level` is evaluated

[12] `http://open-services.net/specifications/requirements-management-2.0/`

3. If the water level fluctuates fast around either value 1 or value 2, then it might be that there is no valve action at all, provided the value is always on the same side of the threshold when it is sampled.

This is usually *not* what would be part of a system design, because it is heavily based on samples that have large time distances. Also, there is no mechanism to prevent use of "implausible" samples, e.g., values that jump up unexpectedly from 0 to 100.

In a way it is a *coarse* model of what the controller is supposed to perform. The test model implies that

- there can be a latency (up to 1s) to a controller reaction,
- it is permissible that the controller reacts very fast,
- it is possible that inputs are ignored if they apply for only a short time (below 1 second).

Based on this, the test model will be used to derive the stimulations to the controller cannot make strong assumptions on how fast and in what manner the SUT will actually react to the inputs. Therefore, the test data generations needs to provide inputs that are stable (e.g., duration 1 second above threshold) in order to ensure that the controller indeed takes a transition.

As a consequence, a large number of perceivable controller implementations will *pass* the criteria of the test model, including one that react immediately to a sample or one that accumulates samples over a short time period (less than 1 second) and reacts according to some *mean value* of the samples. In terms of model-based design, this is a *good thing*.

4 Defining a Test Project in the INTO-CPS Application

Configuring and using a Test Project involves several activities, which include:

- The creation of a test project.
- The definition of tests.
- The compilation of test driver FMUs.
- Setting up test runs.

- Running a test.
- Evaluation of test results.

These activities can be performed using a combination of the INTO-CPS application and the RT-Tester graphical user interface. In the INTO-CPS application test automation functionality can be found below the main activity *TEST-DATA-GENERATION* in its project browser. In this section we keep within the INTO-CPS application interface and give forward reference to RTTUI3[13] usage, which is discussed in Section 5, if appropriate.

Starting the License Management Process Before using most of the test automation utilities, the license management process has to be started. To this, right-click on *TEST-DATA-GENERATION* and select *Start RT-Tester License Dongle* (see Figure 3).

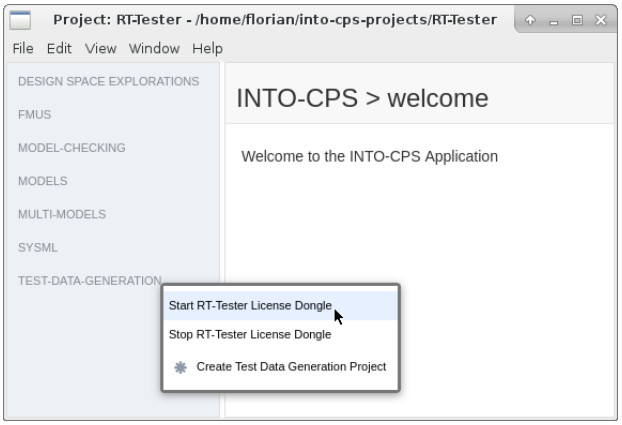

Figure 3: INTO-CPS Application: Starting the License Management Process

Creating a Test Automation Project After having developed the behavioural model in Modelio and exported it to an XMI-file, test automation projects can be created from the INTO-CPS application. Such a project is then added as a sub-project within a containing INTO-CPS application project.

To create a project, right-click on *TEST-DATA-GENERATION* in the project browser and select *Create Test Data Generation Project* (see Figure 4). Then

[13] An older product line with limited capabilities and different look-and-feel was named RTTUI2; "RTTUI3" is often used in documentations to distinguish from that older one.

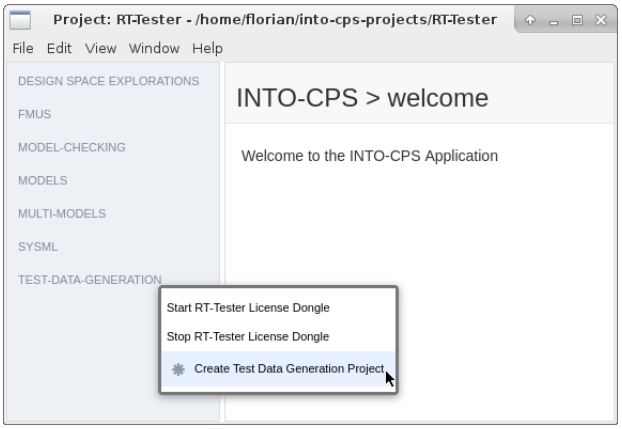

Figure 4: INTO-CPS Application: Create Test Automation Project

specify a name for the project, select the XMI-file with the test-model, and press *create* as in Figure 5. The newly created sub-project and its direc-

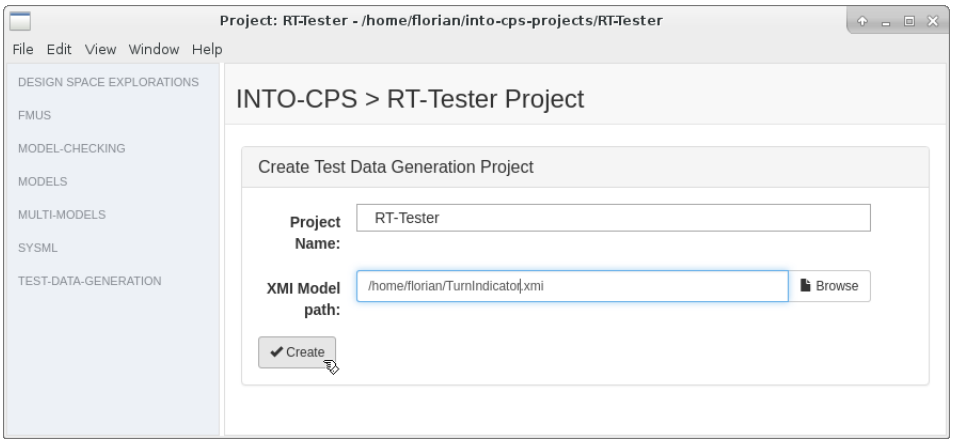

Figure 5: INTO-CPS Application: Create Test Automation Project Dialogue

tory hierarchy is displayed in the project browser. Some directories and files of the RT-Tester project which are not of great importance in the course of INTO-CPS are hidden from the browser. Of special significance are two folders.

1. `TestProcedures` contains symbolic test procedures where test objectives are specified in an abstract way, like for example by specifying LTL formulas.
2. From these symbolic test procedures, concrete executable (RT-Tester 6) test procedures are being generated, which then reside in the folder

`RTT_TestProcedures`.

Defining a test The specification of test objectives is done using the RT-Tester GUI and is explained in more detail in Section 5.2. The relevant files can be opened in the RT-Tester GUI directly from the INTO-CPS application by double-clicking them.

Generating a Test After having defined the test objectives, a concrete test case can be created by a right-click on the symbolic test case under `TestProcedures` and then selecting *Solve* (see Figure 6). A solver com-

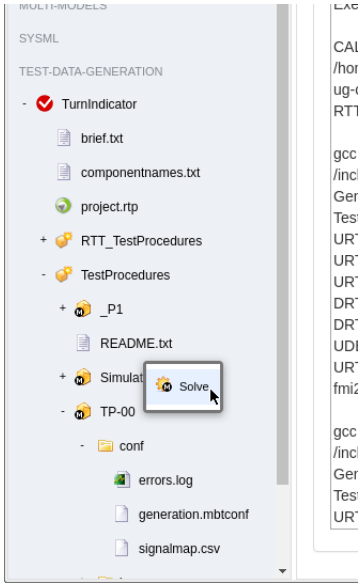

Figure 6: INTO-CPS Application: Generating a Concrete Test Procedure

ponent then computes the necessary timed inputs to realise the test objectives. A concrete test procedure is being generated which feeds a system under test with these inputs and observes its outputs against expected results which are derived from the test model. This test procedure will be placed in `RTT_TestProcedures` and has the same name as the symbolic test procedure. Figure 7 shows the progress of a test generation.

Generating FMUs A generated test procedure can be cast into an FMU which then can be run in a co-simulation against the system under test. To

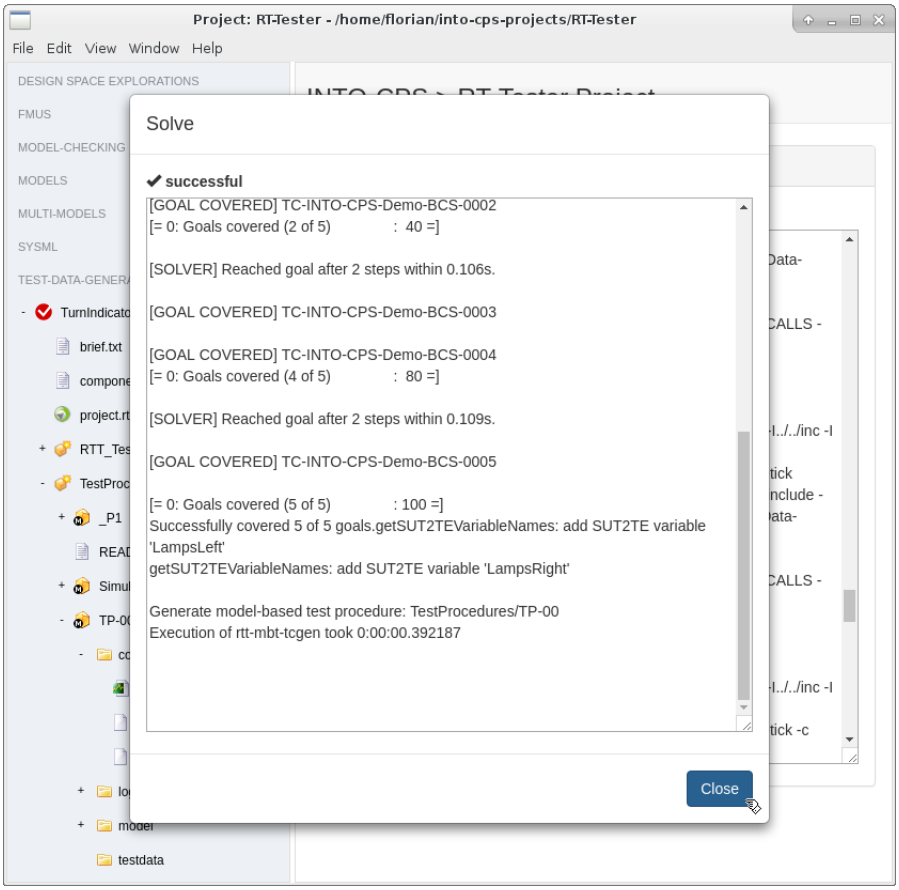

Figure 7: INTO-CPS Application: Progress of test data generation

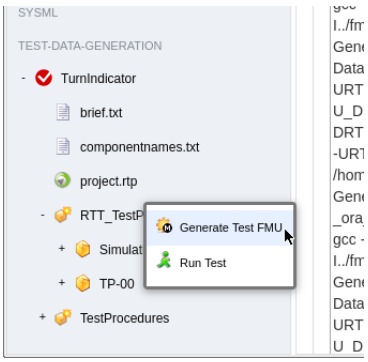

Figure 8: INTO-CPS Application: Generating a test FMU

this end, right click on the concrete test procedure and select *Generate Test FMU* (see Figure 8).

In cases where a real and perhaps physical system under test is not available, a simulation of the system under test can be generated from the behavioural model. To generate such an FMU, right-click on *Simulation* an select *Generate Simulation FMU* as depicted in Figure 9.

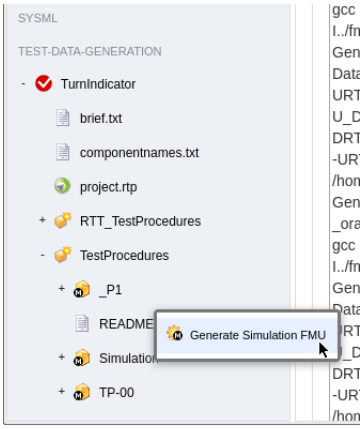

Figure 9: INTO-CPS Application: Generating a simulation FMU

Running a Test In order to run a test, right-click on the test procedure and select *Run Test* (see Figure 10). Then specify the FMU of the system

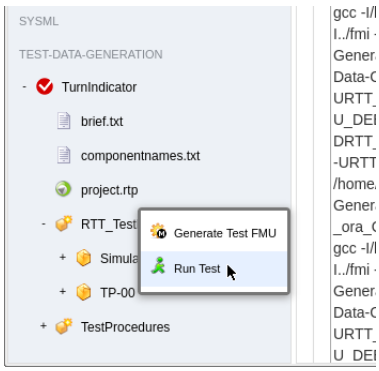

Figure 10: INTO-CPS Application: Start Run-Test dialogue

under test. If the system under test is to be replaced by a simulation, press on the corresponding *Simulation* button. The duration of the test is derived during test data generation and does not need to be manually specified.

However, an appropriate step size must be set. Finally, after making sure the COE is running, press *Run* to start the test (see Figure 11). The next

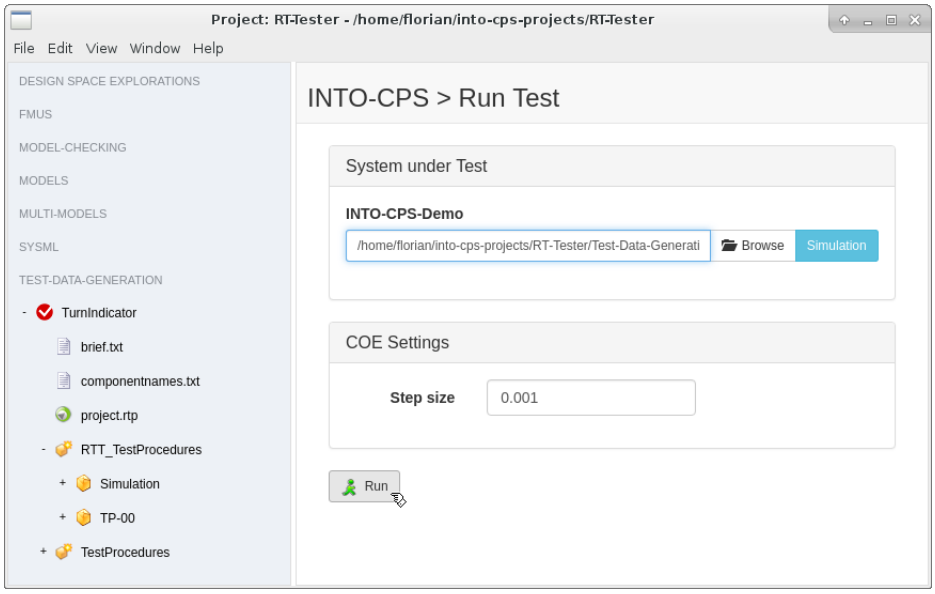

Figure 11: INTO-CPS Application: Run-Test-Dialogue

step is to evaluate the test execution which is described in Section 5.4.

In a realistic project, several test procedures will be defined and run repeatedly. Also, the run has to be evaluated. These details are discussed in the subsequent section.

5 Operating a Test Project with RTTUI3

A test project is based on the elements defined in Section 4. If any of these is changed, all previous results are obsolete and have to be re-iterated in order to give valid information.

The operations of a test project include the following activities.

- Define test goals, which can be translated to test procedures.
- Run test procedures (with the aid of the COE) in order to compare expected (specification) with observed (implemented) behaviour.
- Extend or add test goals in order to increase the number of covered situations.

- Analyse logs of deviations in order to determine whether this is a defect in the specification (test model weakness) or bug in the implementation.

A possible outcome of a test project is the revision of either test model, implementation, or test project generation; in this case, all activities in this section are re-iterated.

A possible completion of a test project is a collection of successful executions of a number of test procedures that in combination achieve the required test depth.

The INTO-CPS Application allows the basic configuration of a test campaign, see Section 4. However, for the more elaborate operations use of the *RT-Tester graphical User Interface (RTTUI3)* is required, since this supports display and editing of all relevant configuration and result files. The screenshots in this section are taken from the RTTUI3.

5.1 The Layout of a Test Campaign

The purpose of a *Test Campaign* is to extend the number of explored situations to the desired level and measure (observe and record) the behaviour of the SUT in all these situation.

The granularity of a situation is also called *Test Case*, which classically consists of a precondition, an (input) event, and an (expected) output.

For model based testing, test cases are derived automatically from the test model by associating model elements with tests case identifiers, compare [Ver15a] for more details. The identifier of each test case include a small shorthand that indicates the nature of the model element, for example:

BCS Basic Control State: corresponds to a basic (non-composite) state of the test model state machine

TR Transition Relation: corresponds to a (possibly guarded) transition in the test model state machine

MCDC Modified Condition/Decision Coverage: corresponds to all situations where *composite transition guards* are evaluated in a set such that all sub-conditions that influence the outcome are evaluated at least once such that it influences the guard value, see [Ver15a] for the full definition.

A *Model-Based Test Procedure*[14] is a container for one or more test cases. Typically these are selected from the automatically derived ones and carry the appropriate shorthand (BCS, TR, MCDC, etc.) in the name. However, the user is free to add arbitrary others in terms of LTL formulae (interpreted over the test model). For model based testing, this configuration is also called *test goal*, because it is attempted to derive a sequence of inputs that will cover all the situations where proper SUT behaviour for all associated test cases is observed.

The model-based test procedures are contained in a folder `TestProcedures`. It is recommended to name them "`TP-...`" or similar, for example `TP-BCS` for one that attempts to cover all basic control states. Based on a model-based test procedure configuration and the test model, the *Solver*[15] attempts to derive a timed trace of input data that covers all the goals. The result is a "concrete" (RT-Tester 6) test procedure (here: also named "`TP-BCS`") that is generated in folder `RTT_TestProcedures`.[16]

The sum of test procedures in a test project allows to make statements concerning the SUT behaviour. In particular a *verdict* is accumulated for every test case, which can be PASS (SUT behaves as expected), FAIL (SUT deviates from expected behaviour), or INCONCLUSIVE (SUT output was not observed). The INCONCLUSIVE case is not necessarily a fault: sometimes there is simply no SUT output that allows to pinpoint whether the behaviour was correct.

As a final step, the test procedure verdicts are mapped back to the requirements in order to determine which parts of the specification have been confirmed to be fulfilled and for which parts deviations have been observed.

5.2 Test Procedure Definition

A (model-based) test procedure is configured by selecting a finite set of test cases that shall be covered by means of providing appropriate input values. In particular **BCS**, **TR**, or **MCDC** can be selected in order to designate all of the test cases of this type (associated to a selected model element) as test goals. In addition to this, arbitrary LTL formulae can be added as goals.

[14] In the RTTUI3, model-based test procedures are marked by icon  .
[15] The Solver is the core component of RTT-MBT, compare [Ver15b].
[16] In the RTTUI3, RT-Tester 6 test procedures are marked by icon  .

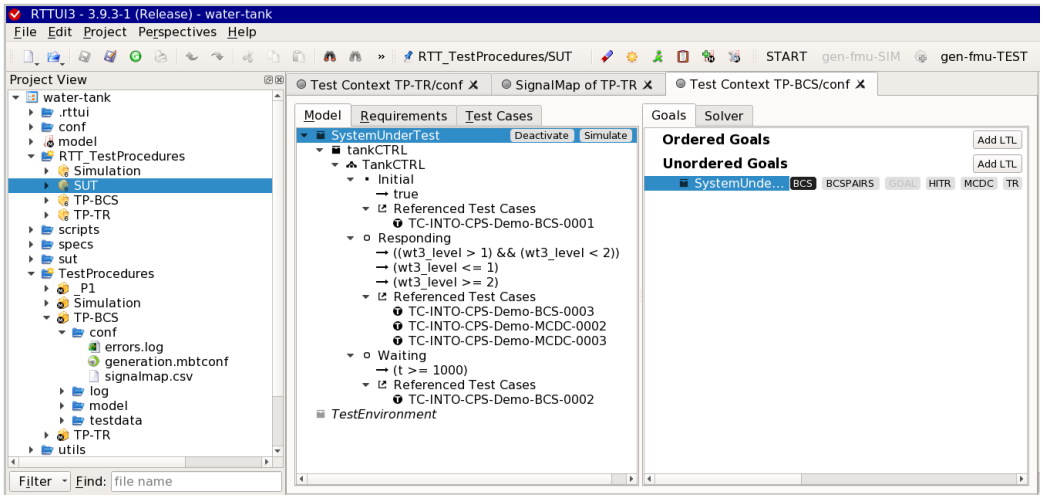

Figure 12: Test Goal: Cover all Basic Control States (BCS).

In Figure 12, the test procedure `TP-BCS` is configured by selecting **BCS** for the SUT.

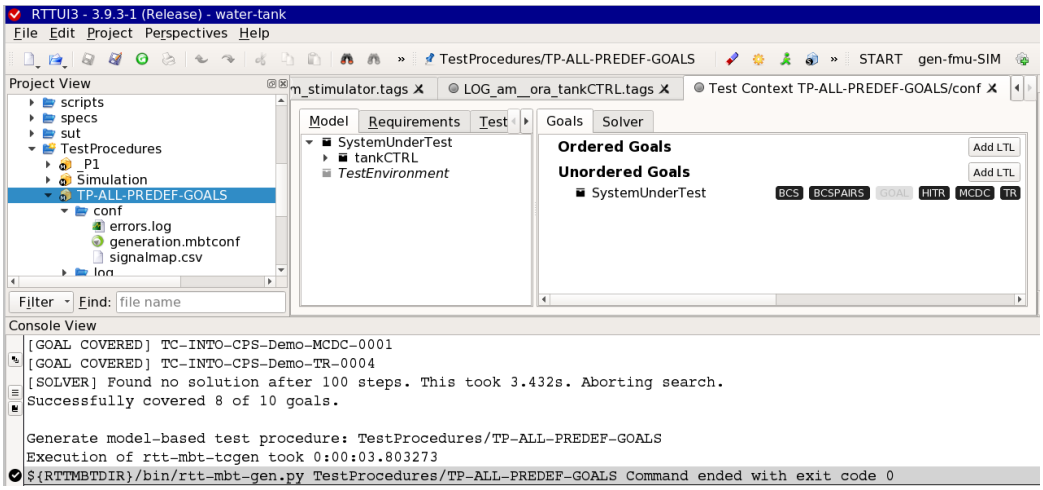

Figure 13: For model-based test procedure `TP-ALL-PREDEF-GOALS`, only 8 of 10 goals are reached by the *Solve* operation.

Not always all goals can be covered. Figure 13 shows what happens for the solve operation (i.e., clicking in the upper right corner) when applied to an overly ambitious test procedure (`TP-ALL-PREDEF-GOALS`). The generated (RT-Tester 6) test procedure only covers 8 of 10 goals. There are several ways to proceed here, including the following.

1. Increase the search depth (in the "`Solver`" tab of the test procedure configuration), per default this is set to 100.
2. Distribute the 10 goals to several test procedures; this simplifies the task of finding a solution.

For every "Solve" step (including those that fail to cover all goals), a test data generation report in HTML format is created.

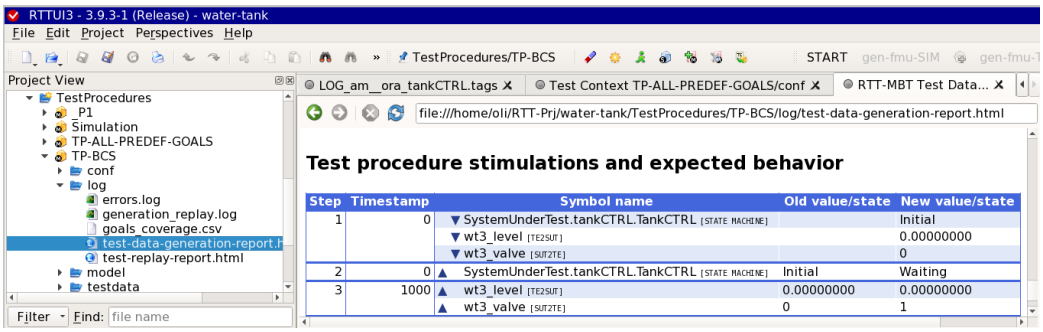

Figure 14: The test data generation report explains the generated inputs and the expected outputs, here for `TP-BCS`.

Figure 14 shows this for the water tank example, where all the basic control states are covered (compare model in Figure 2). In this simple case it suffices to allow time to elapse to the 1000 ms threshold, since this is where the "Waiting" state will be left and the "Responding" state will be reached. The 3rd state is the "Init" state (black bullet), which is covered implicitly.

The water level (SUT input) remains 0 in this run, therefore it is expected from the SUT to close the valve (set `wt3_valve` to 1) at time step 1000.

Configuring Acceptable Latency for SUT Outputs. Not all reactions of the SUT can be assumed to happen instantaneous. Out water tank controller is allowed to take 1000 ms before it reacts on the current water level (compare Figure 2). This means that tests have to be careful with respect to timing assumptions that are made on the SUT implementation.

Part of a test procedure configuration is therefore the definition of allowed *latency* of a SUT output (here: `wt3_valve`). This is shown in Figure 15. Note that the term latency works in both directions of the time line: a SUT implementation that shows the reaction 1050ms *early* as compared to what the test model predicts will also be acceptable.

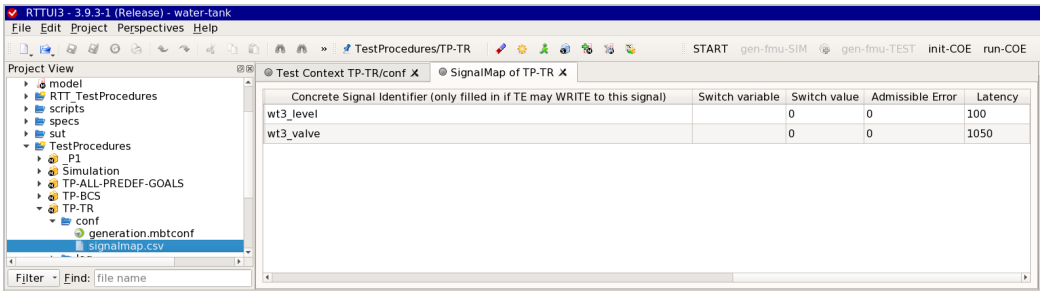

Figure 15: Not all reactions can be assumed to be instantaneous. For the tank controller, the correct `wt3_valve` value may by reached with a *latency*, which is configured here.

If the test designer is satisfied with the results of the "Solve" step, the FMU (which contains the test logic) is generated. This is done by the operation `[gen-fmu-TEST]`, selectable from the command bar. The output of `[gen-fmu-TEST]` is a compiled RT-Tester 6 test procedure with a matching `*.fmu` file. This can be used for executions organised by the COE.

5.3 Test Cases and System Requirements

The concept of a test case is a central term for testing, since it allows to isolate a small part of the system behaviour and associate it with an observation.

However, a test case only lives on the test level (sometimes: verification level) of project. It has to be associated with design or development level objects in order to allow flow up of test results to the development stage.

Typically (as here), test cases are therefore associated with *system requirements*, i.e., organised statements that describe the intended behaviour of the system. In our example, requirements are identified by the shorthand REQ-*xxx*.

Figure 16 shows the relation of test cases to (system) requirements.[17] For example, requirement `REQ-005` is associated with test case `TC-TurnIndication-Controller-BCS-0004`, which designates reaching a specific basic control state.

[17]The screenshot is taken from the "Turn-Indicator" example with an Enterprise Architect export; the requirements export from Modelio is currently not operative.

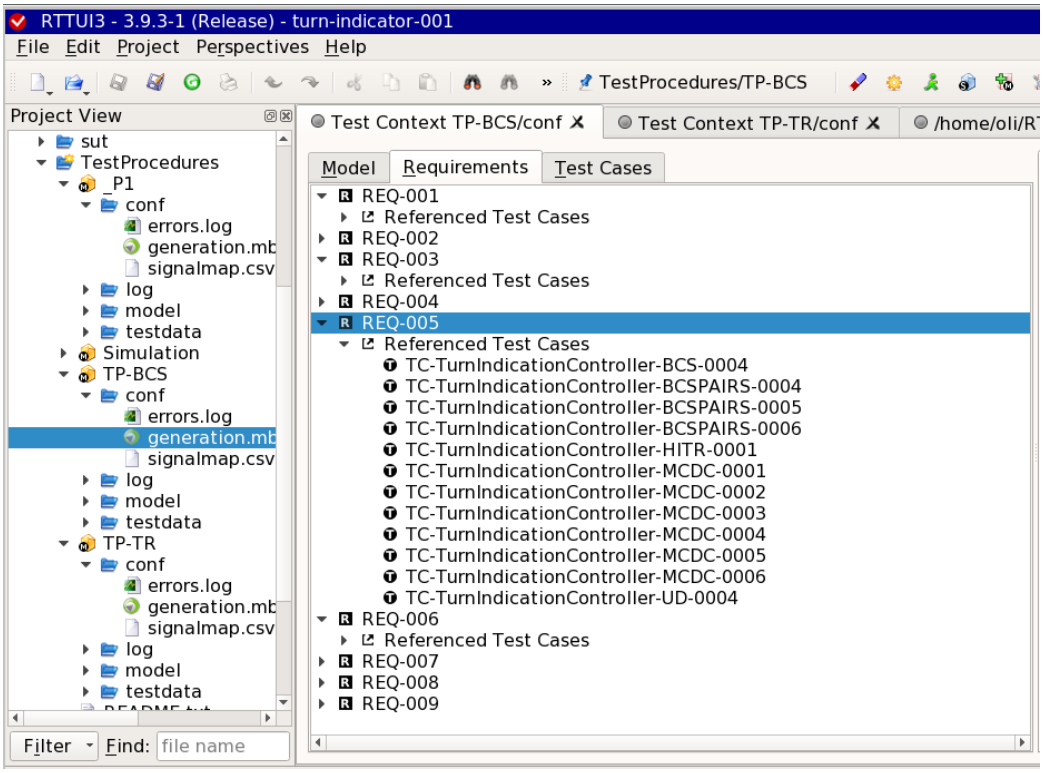

Figure 16: System requirements (REQ-xxx) layed out in the model are associated with test cases (TC-$\star$-$yyyy$).

Since every requirement is associated with at least one test case, the completeness of test cases allows to make a statement (verdict) for every requirement.

5.4 Evaluation of Test Executions

Every test execution (operation `[run-COE]`) yields as result an evaluation of test cases, i.e., associates them with a verdict PASS, FAIL, or INCONCLUSIVE.[18] The details are found in the test log files below testdata, see [Ver15a] for details.

The file `testcase_tags.txt` (Figure 17) gives a condensed record of test case, verdict, and point in a `*.log` file where a corresponding PASS,

[18]The verdict can also be "NOT TESTED": this means a test case has been included in a test procedure but a run that reaches it is still missing.

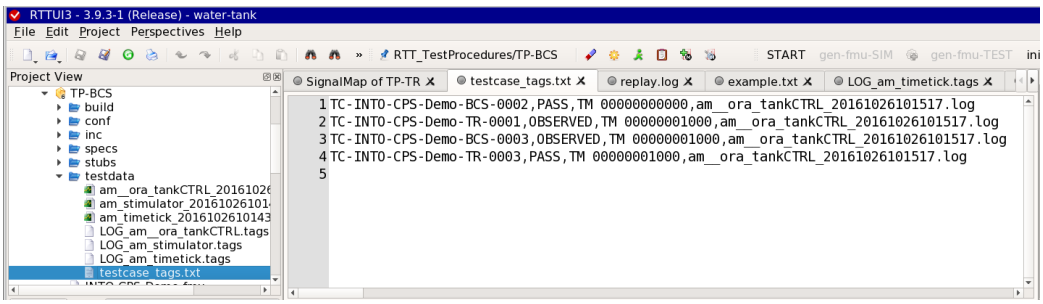

Figure 17: Test Case verdicts from execution of TR-BCS.

FAIL, or—in case of INCONCLUSIVE—test case occurrence without `@rtt-Assert()` can be found.

It should be noted that even though this is recorded as part of the test (here: `TP-BCS`), the verdicts do evaluate in fact the behaviour of the SUT (!) More precisely, they evaluate the behaviour of the FMU that is connected to the test procedure in order to build a closed system with no pending inputs or outputs. Per default, `[run-COE]` will suggest to use "`RTT_TestProcedures/SUT`" as seconds FMU, which in the water tank example runs the code located in the `./sut/` folder. If the code there is incorrect, then some of the tests should fail. In the default installation, we have provided a valid and correct SUT.[19] Other counterparts can be selected for the run, including "`RTT_TestProcedures/Simulation`", which is derived directly from the test model (and thus should behave correctly).

Test Case Verdict Summary (in a Test Project). For a test project, the "sum" of all test case verdicts gives the best overview on the test results (so far). This is also known as *Test Case Coverage*, see Figure 18. A PASS here means that all test procedures that execute that particular test case did yield a pass. A FAIL means that *at least one* test procedure evaluated this test case with FAIL. INCONCLUSIVE means that no test procedure did yield FAIL and at least one test procedure did not put out PASS.

The overview is in the HTML format, and the 'Reference (Origin)' list a hyperlink that allows to jump of the corresponding test log file.

Requirement Verdict Summary (in a Test Project). For the system design/development perspective, the results of the test cases are mapped

[19] This can be modified for experiments, of course.

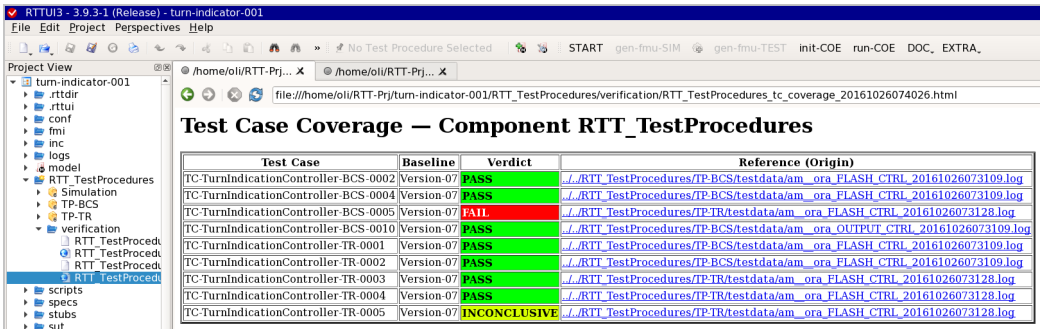

Figure 18: Test Case Tracing, referencing the logs of the tests that contribute to PASS/FAIL/INCONCLUSIVE.

back to the (system) requirements. This allows to see which requirements are observed to be fulfilled by the SUT, and which are not.

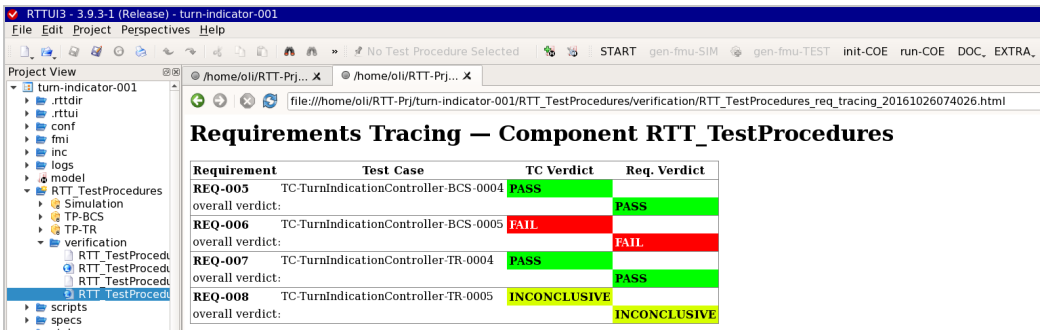

Figure 19: Tracing the Test Case verdicts (PASS/FAIL/INCONCLUSIVE) to the associated requirements.

Figure 19 shows this for a subset of the requirements. In our case, only one test case is associated to each requirement, so the verdict of a requirement is identical with the test case verdict. In general, all associated test cases have to be PASS in order to have a PASSed requirement.

6 Conclusions

In this document we have layed out the test automation operations and capabilities along the lines of a simple water tank controller (defined in Modelio) as a running example. The purpose of the test model has been explained, which not necessarily coincides with a design model. The traceability of requirements has been sketched and will be elaborated during Year3.

The test project configuration from the INTO-CPS Application has been sketched in Section 4.

For the more elaborate configuration, the RT-Tester GUI (RTTUI3) can be used as explained in Section 5. This includes the definition of test goals (based on test cases) and discusses the nature of test results.

More tool capabilities are needed for full integration of system requirements and test result tracing (see limitations listed in Section 2.2), but the provided functionality is sufficient to allow first usage of test automation in user projects, including the pilot studies.

References

[Ver15a] Verified Systems International GmbH, Bremen, Germany. *RT-Tester 6.0: User Manual*, 2015. `https://www.verified.de/products/rt-tester/`, Doc. Id. Verified-INT-014-2003.

[Ver15b] Verified Systems International GmbH, Bremen, Germany. *RT-Tester Model-Based Test Case and Test Data Generator – RTT-MBT: User Manual*, 2015. `https://www.verified.de/products/model-based-testing/`, Doc. Id. Verified-INT-003-2012.

A List of Acronyms

20-sim	Software package for modelling and simulation of dynamic systems
ACA	Automatic Co-model Analysis
ASD	Abstract Syntax Diagram
AU	Aarhus University
BCS	Basic Control States
CLE	ClearSy
CLP	Controllab Products B.V.
COE	Co-simulation Orchestration Engine
CPS	Cyber-Physical Systems
CT	Continuous-Time
DE	Discrete Event
DESTECS	Design Support and Tooling for Embedded Control Software
DSE	Design Space Exploration
FMI	Functional Mockup Interface
FMI-Co	Functional Mockup Interface – for Co-simulation
FMI-ME	Functional Mockup Interface – Model Exchange
FMU	Functional Mockup Unit
HiL	Hardware-in-the-Loop
HMI	Human Machine Interface
HTML	HyperText Markup Language
HW	Hardware
ICT	Information Communication Technology
IDE	Integrated Design Environment
LTL	Linear Time Logic
M&S	Modelling and Simulation
MBD	Model Based Design
MCDC	Modified Condition/Decision Coverage
MiL	Model-in-the-Loop
OMG	Object Management Group
OS	Operating System
OSLC	Open Services for Lifecycle Collaboration
PROV-N	The Provenance Notation
RPC	Remote Procedure Call
RTTUI3	RT-Tester graphical User Interface, sometimes referred to as "`RTTUI`"
SiL	Software-in-the Loop
ST	Softeam
SUT	System Under Test
SVN	Subversion

SysML	Systems Modelling Language
TA	Test Automation
TE	Test Environment
TR	Transition Relations
TRL	Technology Readiness Level
TWT	TWT GmbH Science & Innovation
UML	Unified Modelling Language
UNEW	University of Newcastle upon Tyne
UTP	Unifying Theories of Programming
UTRC	United Technologies Research Center
UY	University of York
VDM	Vienna Development Method
VSI	Verified Systems International
WP	Work Package
XMI	XML Metadata Interchange; here the format that modelling tools uses to exchange model data
XML	Extensible Markup Language