ITEA 3 Call 4: Smart Engineering

D3.6 Test Execution Framework

Project References

Project Acronym	XIVT		
Project Title	eXcellence In Variant Testing		
Project Number	17039		
Project Start Date	November 1, 2018	**Project Duration**	36 months
Project Manager	Gunnar Widforss, Bombardier Transportation, Sweden		
Website	https://www.xivt.org/		

Document References

Work Package	WP 3: Testing of Configurable Products	
Task	T3.3: Test Execution Framework	
Version	V 1.0	Oct 31st, 2020
Deliverable Type	R (Report)	
Dissemination Level	Software: CO (Confidential) Only for members of the Consortium. Documentation: Public	

Summary

Building on results from previous and ongoing projects as well as on publicly available solutions, a test execution framework is being developed, in which test cases for variants can be executed both in software-in-the-loop (simulated distributed software systems) and hardware-in-the-loop (embedded controllers). In this document, an overview of the targeted use cases is given and the installation and use of the test execution framework is explained.

Table of Contents

1. Introduction

Building on results from previous and ongoing projects as well as on publicly available solutions such as, e.g., the Robot Framework for test automation and RPA, the Robot Operating System ROS, the Gazebo simulator and others, a test execution framework is being developed, in which test cases for variant and configurable robotic applications can be executed both in software-in-the-loop (simulated distributed software systems) and hardware-in-the-loop (embedded controllers).

The development of the test execution framework follows an incremental approach. In its initial version, the requirements are mostly taken from a single selected use case. However, the design is carried out with generalization and extensibility in mind. During the further development, requirements from other use cases and desirable tool integrations are taken into account.

In the following sections, an overview of the targeted use cases is given and another existing solution for test execution in a similar but slightly different context, developed by a project partner, is presented.

1.1. Targeted Use Cases

1.1.1. FFT CUBE Demonstrator / FOKUS Robotics Demonstrator

FFT's Test Process for new components is right now not complete and good as possible. The first step for each usage of a new component in a new production cell, like electric drive, is that FFT orders one real hardware. With the added documentation a signal trace is designed. The next step would be to create a PLC block for the component. After the implementation the PLC block will be tested against the hardware component on the construction side or in the technology lab of FFT. If changes are needed they are directly implemented. The goal is that the component can work properly like described in the documentation of the component manufacturer. After this test on component level, tests are performed in combination with other components, like a total turntable that's installed on the electric rotary drive. Each component can be replaced by a similar one from another manufacturer. But there are no regression tests performed for old variants. Therefore it is possible that some errors will occur later on when there is a replacement and these are not taken into account at all.

The FOKUS Robotics Demonstrator is an effort to make the challenges from the FFT CUBE Demonstrator use case more tangible for research by providing a downscaled version. This is achieved through the use of affordable hardware and open-source software. The demonstrator will enable researchers and solution providers to work on the underlying challenges of the use case without taking into account many of the technicalities involved in working with industrial robots. In order to ensure the applicability of the results to the actual use case, the demonstrator is being developed in close collaboration with the original use case provider.

1.1.2. Advanced Driver Assistance Systems: Pedestrian Detection System

The test execution aspect of this use case is still in its early stages. Testing is performed entirely manual and as of now, there is no formal testing process specified.

1.1.3. Propulsion and Controls (PPC)

Testing at Bombardier is performed in three different phases and at four different levels as shown in Figure BTTest. In order to achieve a Safety Integrity Level (SIL) 2 compliance for the propulsion software, testing on some test levels has to conform to the EN50128 or EN50657 standard for development of rail control software. Note that the restrictions placed on the test process only apply to some levels and not all. Specifically, the standard applies to hardware-software integration, software integration, and components.

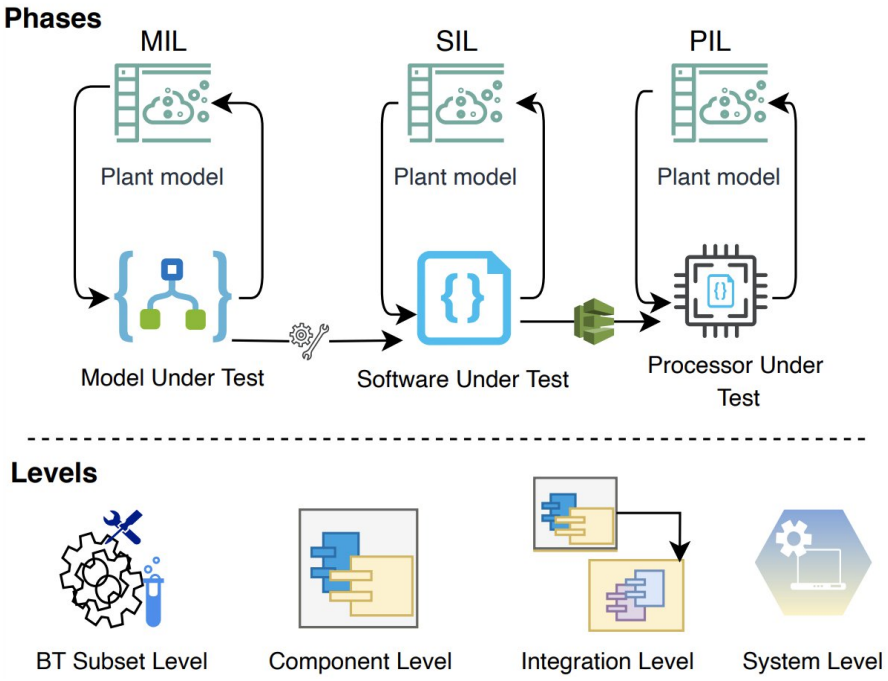

Figure BTTest: Testing at Bombardier's Propulsion Control Team

The developed artifacts for the derived products (Simulink models) are tested in-house in three different phases. If an artifact has any hardware/environment dependencies, a plant model is created to simulate the required behavior of the hardware/environment. The artifact is the first tested in a Model-In-the-Loop (MIL) environment inside Simulink. Embedded Coder is then used to generate C/C++ code from the Simulink model. The generated code is then tested in a Software-In-the-Loop (SIL) environment. For safety-critical components, the generated code is also subjected to code reviews. The reviewed code is deployed on the target processor for a Processor-In-the-Loop (PIL) testing.

All the tests are created as Simulink Test Harnesses and are managed and executed via Simulink Test Manager. In execution, independent test cases are executed in parallel and

coverage is recorded. Finally, test coverage reports are generated containing coverage reports for the software and requirement.

These phases are performed on four different levels. First (unit level), the library is tested, and the source code is reviewed. Then the components (created from combining the sub-set elements) are subjected to all phases of the testing. Thirdly, the components are integrated in a way that serves a system function. The integrated components are tested in all three phases for integration errors. Lastly, the propulsion system is tested in all three phases at the system level. In addition, a final Hardware-In-the-Loop (HIL) testing is performed off-site. Note that the SPL itself is only tested at the BT subset level and partially at the component level. Components that are integration dependent are only tested in the derived products.

1.1.4. Raw Map Data in Autonomous Vehicles

The overall testing process is depicted in Figure 1. It is verification against the original map database. We need to verify the database in the compressed form represents the original data we've got from the supplier. The reason for this verification is due to missing data (some areas might not be mapped, e.g. due to Occlusion), having new construction zones, newly added roads, etc. The supplier provides High-definition map data by utilizing a very accurate and expensive set of sensors every three months; however, they are planning to provide the database monthly.

After receiving the map database from a supplier which have a very large size, GM compresses the database. This process takes 2 weeks. Then GM needs to verify the compressed database against the original database (First testing phase). This testing verification process takes another two weeks to fully (%100) verify the database (using the brute-force algorithm). In this testing process, the map routes are selected to verify the database can be considered as test cases.

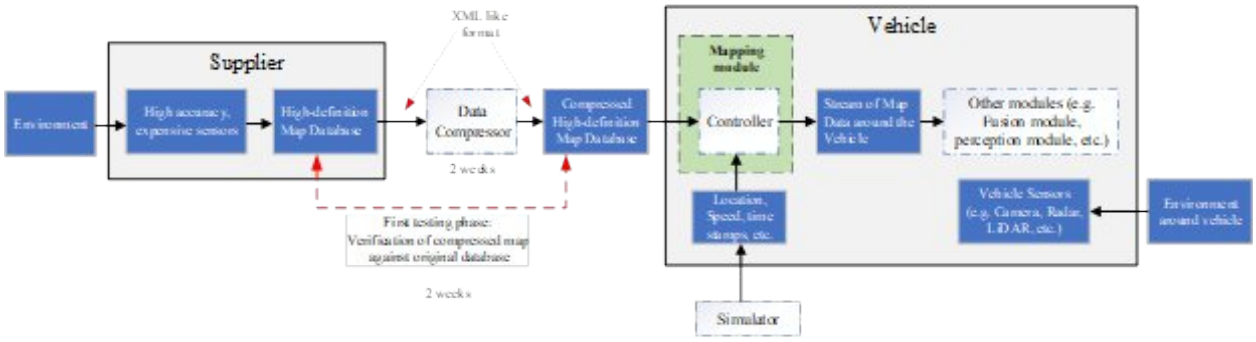

Figure 1. Overall system setup representing two testing phases.

Current testing method

1. Driving a real car and testing the map database
2. Simulating the car driving around and checking the map database

Current testing (linked) limitations

1. Testing was performed on the HW module and limited to real-time testing. There is a certain bandwidth limit that the HW module can consume and process the input data (road segment).

2. There is no intelligence in the current process of map coverage. Test engineers are doing is a brute-force algorithm. They select a random road in the database and drive in mostly straight lines until they reach the end of the road and keep doing that until reaching an appropriate level of coverage.

1.2. Existing Solutions

At ifak, a tool for test execution has been developed which is integrated in ifak's model-based testing tool chain. It connects the previous steps of test generation, test management and test prioritization with the System Under Test (SUT). Test execution enables the tool chain to connect to different systems and execute the generated test cases.

The execution of the generated test cases is enabled via a protocol-independent interface for connecting the test tool used to the SUT as hardware or software. Since these interfaces are often very heterogeneous, especially in industrial automation, a protocol-independent way to connect the test suite, a test adapter, is required. This adapter abstracts the communication process from the test system and provides it with a uniform interface. Using various communication protocols such as OPC UA or Modbus, the internal test communication from and to the SUT is realized. For the SUT, the test scenario therefore does not differ from the integration into a real environment. The SUT is exposed to specific communication conditions and scenarios just like in a real environment without any special preparation.

Therefore, a test adapter was designed, which allows to break down the essential features of a test execution to a common set of actions with a common language. In combination with a configuration that maps the system specifics, the communication with the different SUTs can be realized and the test tool chain can be completed.

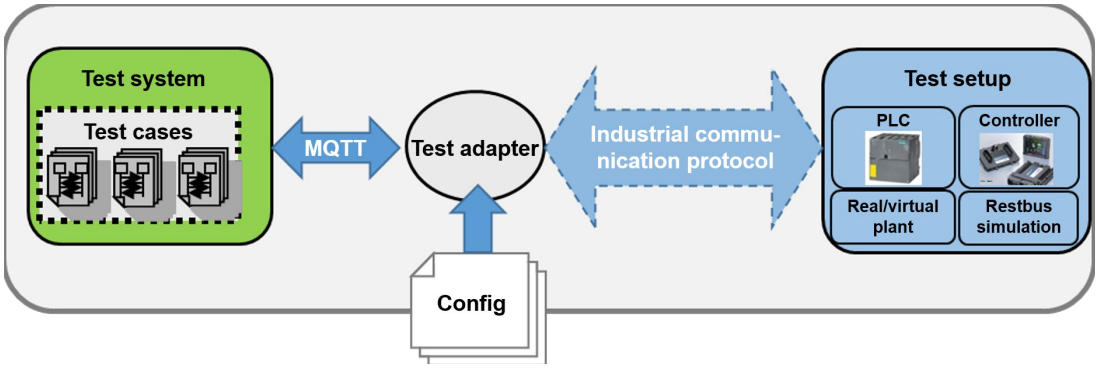

Figure ifakTestExecutor: The modular test adapter as link between test system and SUT

The test execution procedure is shown in the Figure ifakTestExecutor. The tool chain for test automation comprises a test system in which the test cases to be executed are defined. For the execution of the test cases and thus for the communication with the SUT the modular test adapter is used. The test system communicates with the test adapter via a clearly defined communication protocol. This is the protocol standard MQTT defined by OASIS in version 5.0

[OA19]. Based on a configuration, the test adapter is then able to transform the test step sent out by the test system into a format and communication protocol (e.g. OPC UA, Modbus, UDP) that is understandable for the SUT. Conversely, the test adapter can also receive messages from the SUT and forward them to the test system.

In order to realize a uniform language for test execution, essentially every sub-step of test execution must be mapped to a common language element. These sub-steps basically perform an atomic action on the system under test. Such sub-steps are for example "Read sensor value X", "move to position Y" or "Stop".

If such test steps are considered in detail, an action that a system must perform can be mapped to three types of actions as part of a test execution. Executing a function, reading values and writing values. Although the respective steps required to do this are heterogeneous and complex, this part of the execution is transparent to the test tool chain. Reducing the chain to these three basic functions considerably simplifies communication on the test generation side, since the tool chain only needs to know these operations.

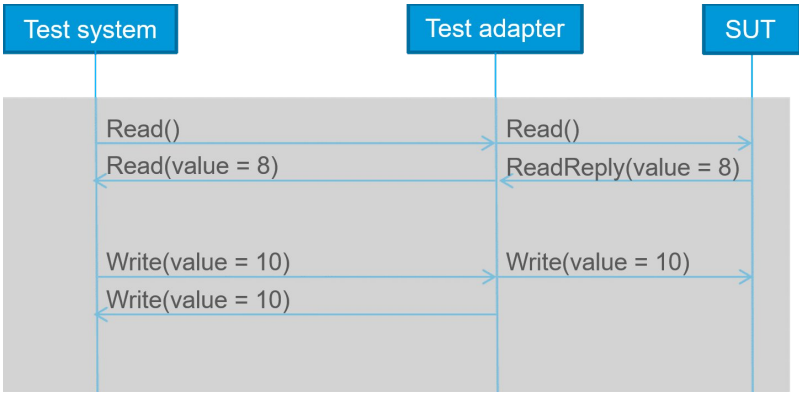

Figure Example 1: Example test execution

By mapping to these basic actions the basic communication with the SUT can be realized. However, some additional elements are necessary to make test generation independent from test execution. For example, a test on which a component should perform a certain action requires a time X. This time depends on the components and can vary depending on hardware or software control. This cannot be solved trivially by a simple general configuration and a language that can essentially read, write and execute. Although a test case can be generated, which writes the value X into register X1, the test case should also check afterwards, whether the target action has been executed successfully. In the theoretical view, this function would be sufficient to execute test cases. However, physical systems often need time to perform certain actions. This time is rarely exactly predictable and often deliberately variable. Furthermore, these time intervals are test system specific and have to be mapped differently in the test case from system to system.

Figure Example 2: Example OPC-UA Code

To illustrate the problem, the above example is used again. We write the value X into a register X1, which shall trigger a move command. Then it shall be checked, if the position is reached. If the value of the position-register would be checked directly after the move-command, it would nearly always fail, because the SUT is still in motion. To map this physical behavior, further elements like a timeout and expected values are required. Looking at the test case in terms of content, it would specify something like: "The SUT must have reached the target position X after n seconds after a move command at position X". The test adapter can map exactly this. After writing a value X, an expected value and a timeout can be specified during reading. The adapter will then periodically check the value or wait for a value change, depending on the protocol behind it. If it reaches this value within the n seconds, it will be reported back. If it does not reach it, the last known incorrect value is returned. The following graphics show the different communication scenarios. Primarily the three basic scenarios Read, Write, Execute are shown. The action for reading can be configured with timeouts and the expected value to be able to map different scenarios. Either a direct response from the SUT can be expected without timeouts, in which case the current value is returned. If the test case expects an event to occur within a time period, timeout and expected value are specified. The write and execution actions do not expect any further parameters, because it can be assumed that write actions and executions can occur at any time, or a test case normally does not expect to set a value. Although there may be test cases which require the checking of the value when setting, this would have to be mapped explicitly by setting and then checking or reading, since only the successful execution of the write command does not explicitly indicate the successful setting of a value. Especially for communication tests this cannot be assumed.

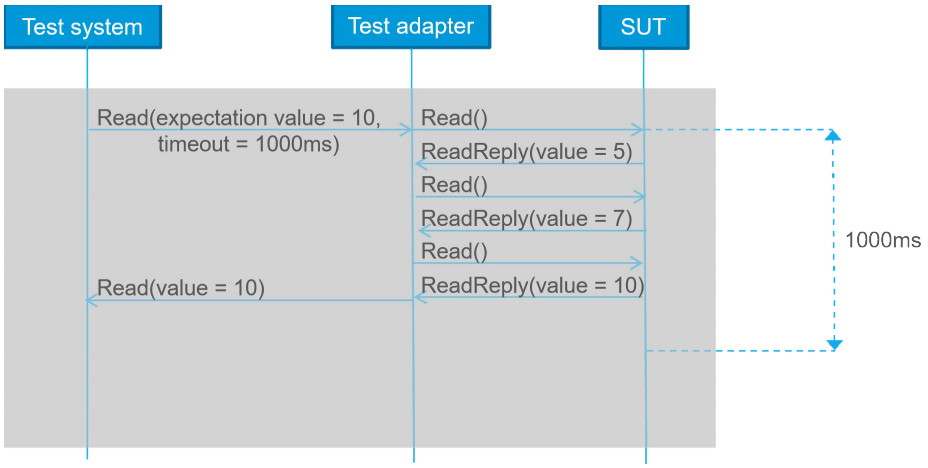

Figure Example 3: Example test execution

1.3. Definitions, Acronyms, Abbreviations

MQTT, Message Queuing Telemetry Transport, an open OASIS and ISO standard (ISO/IEC 20922) lightweight, publish-subscribe network protocol that transports messages between devices. See https://mqtt.org/

PLC, Programmable Logic Controller

ROS, Robot Operation System, “The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.” See https://www.ros.org/about-ros/

ROS-Industrial, “ROS-Industrial is an open-source project that extends the advanced capabilities of ROS software to industrial relevant hardware and applications.” See https://rosindustrial.org/

ROSin Project, See https://www.rosin-project.eu/

RPA, Robotic Process Automation

SPL, Software Product Line

SUT, System under test, aka. test item

2. Technical and User Documentation

The implementation of the test execution framework is based on the Robot Framework. The Robot Framework is a widely adopted open-source automation framework for test automation as well as RPA. It is domain and language agnostic and allows for extension through plugins.

The central component of the test execution framework is a plugin that provides generic keywords for interaction with ROS and Gazebo based robotics applications. These generic keywords can then be combined into use case specific keywords or directly be used in writing test cases.

2.1. Installation

The installation is tested to work with Ubuntu 18.04, Python 3, ROS Melodic and Gazebo 9. In order to install the test execution framework, the following steps are needed.

Create a working directory for your project:

```
~$ mkdir example_project && cd example_project
```

Optional: Create and activate a virtual Python environment inside your working directory:

```
example_project$ python3 -m venv .venv
example_project$ source .venv/bin/activate
```

Install Robot Framework. For more detail, see also the Robot Framework Installation instructions[1]:

```
(.venv) example_project$ pip install robotframework
```

Download and install the text execution framework from GitLab:

```
(.venv) example_project$ git clone \
> git@gitlab.com:xivt/itea/test-execution-framework.git
(.venv) example_project$ pip install \
> ./test-execution-framework/robot-ros-gazebo-library
```

Optional: Install `catkin_manager` from GitLab:

```
(.venv) example_project$ pip install \
> ./test-execution-framework/catkin_manager/
```

The installation is now complete.

[1] Available from https://github.com/robotframework/robotframework/blob/master/INSTALL.rst

2.2. Writing Test Cases

In the following, we are giving a brief introduction on writing test cases with the Robot Framework in general and the ROS and Gazebo specific keywords in particular. For a detailed introduction to writing test cases with the Robot Framework, please refer to section 2 of the Robot Framework User Guide[2].

Test suites and test cases for the Robot Framework can be written in different formats. For this introduction, we chose the so-called space separated format. A simple test suite may consist of a single file containing one or more test cases and each test case consists of a series of keywords. Keywords serve as an abstraction for underlying functionality, which either stimulates the SUT or verifies responses from or properties of the SUT. Besides a set of basic keywords available from the Robot Framework itself, most keywords are provided by third-party plugins serving as keyword libraries.

An example test suite using the ROS and Gazebo specific keywords may look like this:

```
*** Settings ***
Documentation     An example test suite
Library           RobotRosGazeboLibrary.Keywords

*** Test Cases ***
Test Handover
    [Setup]    Connect on port 9090
    Launch xivt_robotics_demo focus_cube.launch
    Verify link robot1::base_link at 0 0 0
    Verify model object at 0.3 0 0
    Run  test_move test_case_1
    Wait for 45
    Verify model object at 0.3 0.63 0
    [Teardown]    Disconnect from ROS
```

The following keywords are available:

Connect on port PORT: Establishes a connection with the ROS master node. This is required before any other ROS specific keyword can be used.

Launch ROS_PACKAGE LAUNCH_FILE: Launches a launch file from the specified ROS package using `roslaunch`.

Run ROS_PACKAGE EXECUTABLE: Runs an executable from the specified ROS package using `rosrun`.

Verify model MODEL_NAME at X Y Z: Compares the actual (simulated) position of a model with the specified expected position. Fails if the deviation is above a certain threshold. In the context of ROS and Gazebo, robots and other objects are represented as models. This keyword is especially useful to test for the position of a model as a whole. An obvious

[2] Available at https://robotframework.org/robotframework/3.2.2/RobotFrameworkUserGuide.html

application would be for example to test for the position of a movable object which is being manipulated by a robotic arm.

Verify link LINK_NAME at X Y Z: Compares the actual (simulated) position of a link with the specified expected position. Fails if the deviation is above a certain threshold. In the context of ROS and Gazebo, models of robots and other objects consist of links and joints. This keyword is especially useful to test for the position of a part of a robot in contrast to the robot as a whole. An obvious application would be for example to test for the so-called tool center point (TCP) link, which serves as a point of reference for the path planning of a robots end effector.

Wait for DURATION: Pauses the execution of the test script for the specified duration based on simulation time.

Disconnect from ROS: Disconnects from the ROS master node and stops all running processes that were previously started using the “Launch” or “Run” keyword.

2.3. Executing Test Cases

In the following, we assume you have a catkin workspace inside your working directory:

```
example_project/ # working directory
    .venv/ ... # virtual Python environment
    catkin_ws/ # catkin workspace
        src/
            example_package/ # your ROS package
                launch/
                    example.launch # your launch file
                ...
    test-execution-framework/ ... # downloaded from GitLab
    example.robot # your test suite
```

The followings three steps each need to be executed in a separate terminal window.

Launch the simulation:

```
example_project$ cd catkin_ws/
example_project/catkin_ws$ source /opt/ros/$ROS_DISTRO/setup.bash
example_project/catkin_ws$ source devel/setup.bash
example_project/catkin_ws$ roslaunch example_package example.launch
```

Start ROS bridge server:

```
example_project$ cd catkin_ws/
example_project/catkin_ws$ source /opt/ros/$ROS_DISTRO/setup.bash
example_project/catkin_ws$ source devel/setup.bash
example_project/catkin_ws$ roslaunch rosbridge_server \
> rosbridge_websocket.launch
```

Run the test suite:

```
example_project$ source .venv/bin/activate
(.venv) example_project$ robot example.robot
```

2.4. Using catkin_manager

The catkin_manager tool is an optional addition to the test execution framework, designed to solve a challenge in dependency management. When working with ROS packages from different sources, the catkin workspace quickly becomes a mess. Some possible reasons for this are:

- Working with different forks of a package results in name clashes
- Heterogeneity of repository structures, e.g.
  - one package per repository
  - multiple packages per repository
  - repositories containing their own catkin workspace
  - etc.

Also, using the same packages in different catkin workspaces requires to have a copy of the code in each of them, leading to code duplication.

Instead of cloning repositories into the catkin workspace, we suggest to create symbolic links from the catkin workspace to the location of the respective ROS packages. The catkin_manager tool supports the management of such links.

The tool expects a file named `ros_packages.yaml` inside the catkin workspace:

```
catkin_ws/
    build/
    devel/
    src/
    ros_packages.yaml
```

The file contains a mapping between package names to file system locations:

```
packages:
    xivt_robotics_demo: ../ros-packages/xivt_robotics_demo
    present_other_location: ../../examples/present_other_location
    required_but_missing: ../../examples/required_but_missing
```

The tool is then used as follows.

To print information about the packages listed in `ros_packages.yaml` and the respective links:

```
(.venv) catkin_ws$ catkin_manager info
[Installed] present_not_required
[Different] present_other_location
[Missing] required_but_missing
[OK] xivt_robotics_demo
(.venv) catkin_ws$ catkin_manager.py info --verbose
[Installed] present_not_required -> ../../examples/present_not_required
[Different] present_other_location
-> ../../examples_other/present_other_location
```

```
[Missing] required_but_missing -> ../../examples/required_but_missing
[OK] xivt_robotics_demo -> ../ros-packages/xivt_robotics_demo
```

To create the links as specified in `ros_packages.yaml`:

```
(.venv) catkin_ws$ catkin_manager link
ln -s ../../examples/present_other_location present_other_location
ln -s ../../examples/required_but_missing required_but_missing
```

To also remove links that are not specifed in `ros_packages.yaml`:

```
(.venv) catkin_ws$ catkin_manager link -d
ln -s ../../examples/present_other_location present_other_location
ln -s ../../examples/required_but_missing required_but_missing
rm present_not_required
```

3. References

[XVT19] XIVT Project Consortium, “XIVT Full Project Proposal Annex.” 03 August 2019.

[OA19] OASIS: MQTT Version 5.0, 2019.

Appendix A - Tool Releases

Tool releases are hosted at https://gitlab.com/xivt/itea. Repositories can be accessed using the following credentials:

- **Username:** `ITEA3XIVT`
- **Password:** `20222018XIVT`