AUTOMATED VERIFICATION ENVIRONMENT FOR TWINCAT PLC PROGRAMS

A. Beckmann, European XFEL, Hamburg, Germany

Abstract

The European XFEL will have three undulator systems SASE1, SASE2, and SASE3 to produce extremely brilliant, ultra-short pulses of x-rays with wavelengths down to 0.1 nm. The undulator gap is adjustable in order to vary photon beam energy. The corresponding motion control is implemented with industrial PCs running Beckhoff TwinCAT Programmable Logic Controllers (PLCs). So far, the functionality of the PLC programs has been verified on system level with the final hardware. This is a time-consuming manual task, but may also damage the hardware in case of severe program failures. To improve the verification process of PLC programs, a test environment with simulated hardware has been set up. It uses a virtual machine to run the PLC program together with a verification program that simulates the behaviour of the hardware. Test execution and result checking is automated with the help of scripts, which communicate with the verification program to stimulate the PLC program. Thus, functional verification of PLC programs is reduced to running a set of scripts, without the need to connect to real hardware and without manual effort.

BACKGROUND

In automation, PLC programs are used to control devices with actuators based on feedback from sensors. The PLC runs a loop: first sensor values are applied to the program inputs, next the PLC program is executed to calculate the output based on the values from the input, and finally the program output is applied to the actuators. PLCs also provide network access to internal memory and variables in order to modify the behaviour of the program and to read out state information. The PLC is supported by a Numeric Control (NC) to control the positioning of movable axes.

The control of the European XFEL undulator systems is implemented using Beckhoff TwinCAT, which offers real-time PLC/NC on Windows based PCs. The PLC/NC is connected to the sensors and actuators of the undulator device via an EtherCAT field bus.

Access from the network side is provided by the Automation Device Specification (ADS) protocol. The protocol is disclosed to the public, but only the .NET library is officially supported by Beckhoff. This basically limits the development of software using the ADS protocol to the Windows platform.

PLC PROGRAM VERIFICATION

The functional verification of the PLC program is an important step within the development process in order to ensure proper operation. In the past, the program was verified in the field by installing it on the device and testing manually. The disadvantages are obvious: due to the high effort only the functionality of the modified part of the program is tested, and failures may lead to severe damages of the hardware device.

To improve the verification process, an automated verification environment has been developed. It allows running a set of tests using a single script. Each test is self-checking, i.e. it decides itself whether a test has passed or failed. If all tests are run and they all pass, then the functionality of the PLC program is said to be fully verified.

Instead of connecting to a real hardware device, the inputs and outputs of the PLC program under verification (PUV) are connected to a Simulation PLC, which simulates the behaviour of the hardware device. Thus, verification can be done in an office environment, and in addition, the risk of hardware damages during verification is eliminated.

Verification Environment Architecture

Testcases are implemented on top of a verification environment, as shown in Fig. 1. The environment is built from verification components (VC) that abstracts the details of communication with the PLC subsystem from the testcases.

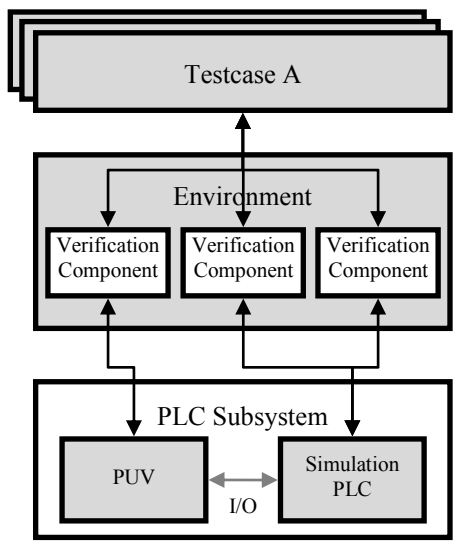

Figure 1: Verification Environment Architecture.

Each VC covers a specific interface of the PUV, which is either a software interface or a physical interface. A software interface is a set of internal variables, and the VC can access them directly via ADS. A physical interface is a collection of I/O signals, and the VC can

ISBN 978-3-95450-139-7

drive them by accessing the Simulation PLC via ADS. A PUV has usually one software interface and several physical interfaces, each for a specific subset of the I/Os.

Towards the testcases, verification components provide a logical interface at a higher level of abstraction, which is called transaction level. A transaction is a single transfer of high-level data, for example the information to move a motor to a new position. Within the VC, the transaction is translated into a sequence of accesses to the PUV in order to move the motor to the new position. The use of transactions facilitates the development of testcases, since they allow defining scenarios on a higher level of abstraction without the need to know the exact details of driving individual PUV interfaces.

A testcase defines a scenario as a sequence of transactions, which are sent to the VCs instantiated inside the environment. At the end, results are verified against expected values using assertions. A testcase passes only, if all assertions are true. The self-checking of results within the testcase is important, since it allows running the test automatically with a final pass/fail decision.

PLC Verification Library

The creation of verification environments and testcases is facilitated by using common structures, which are bundled into a PLC Verification Library (PVL). It is implemented as a Python package named *pvl*, containing subpackages *core*, *util*, and *vc* to further divide the code, as shown in Fig. 2.

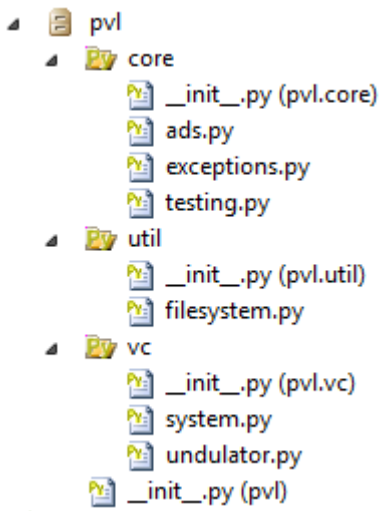

Figure 2: PVL Package Hierarchy.

The *core* subpackage contains the ADS communication class, some exception classes used internally, a set of base classes to build testcases and environments, and the test suite class.

The ADS communication class *Ads* provides methods to read and write PLC variables of different type either by name or by address. It is a facade to the TwinCAT ADS library from Beckhoff to provide a simplified interface to the PLC for the verification environment and its components. Exceptions raised inside the ADS library are catched and rethrown as new PVL exceptions.

A project specific verification environment is a class derived from the *TestEnvBase* class. In the constructor, it instantiates the verification components required to interface to the PLCs. It may override a *setup()* and *teardown()* method, derived from *TestEnvBase*, to perform some initialisation and clean-up before and after a testcase. *TestEnvBase* also contains a reference to the ADS communication class, which is used by the VCs to exchange data with the PLCs.

The project specific testcases are classes derived from the *TestBase* class. Within the testcase class, the scenario is implemented by overwriting the *run()* method of the base class. This method is called from PVL to execute the testcase. *run()* takes one argument to pass in the environment, so that it can be accessed from within the scenario. The *TestBase* class provides methods to make assertions, which are used in the *run()* method to verify results from the scenario. A failed assertion immediately stops the execution of the scenario and marks the testcase as failed.

The *TestSuite* class handles the execution of a set of test cases. It instantiates the ADS communication class, the project verification environment, and all testcases. At the beginning, it opens an ADS connection to the PLCs, then for each testcase, it calls the *setup()* method of the environment, then the *run()* method of the testcase, and afterwards the *teardown()* method of the environment. At the end, it closes the ADS connection to the PLCs, and generates a report about the testcase results.

The *util* subpackage contains some methods to collect names of files, and to check the current working directory. They are used only by the main simulation script to find out the file locations of testcases and the verification environment, and whether the script is started from the right directory. This is important, since the script relies on a specific directory layout to find files.

The *vc* subpackage contains verification components to interface to the TwinCAT system and also to specific PLC programs, such as used for the undulator control system. The system VCs provides methods to start, stop and reset a PLC, or to modify NC parameters of simulation axes.

Verification components use the ADS communication class from the *core* subpackage to communicate with the PLC system. The reference to that class is stored inside the verification environment and is passed as an argument to the constructor of each VC.

PLC Subsystem

The PLC subsystem is configured to run two PLCs in parallel: one for the PUV and one for the simulation program. Beckhoff TwinCAT 2.11 provides up to four virtual “PLC CPUs” to run multiple PLCs on one physical system. The PUV is loaded onto virtual CPU #1, and the simulation program is loaded onto virtual CPU #2.

The PLC subsystem configuration is different from the real system configuration. Inputs of the PUV are directly

ISBN 978-3-95450-139-7

linked to corresponding outputs of the simulation program, and outputs of the PUV are linked to corresponding inputs of the simulation program. The movable axes of the real hardware device are replaced by simulation axes.

The main task of the simulation program is to drive the inputs of the PUV similar to the real hardware device. An example is the triggering of limit switch inputs, if the undulator gap is reaching either its minimum or maximum value. In addition, the simulation program provides ADS access for the VCs to the PUV I/Os.

Scripting

The main script to run a testcases is a batch file named *plcsim.bat* that can be directly executed from the command shell. Its only purpose is to set the environment variable IRONPYTHONPATH to the list of directories, where PVL and the ADS .NET library are installed, and then call the Python interpreter with the file *plcsim.py*.

plcsim.py first locates the file containing the verification environment, and then the files containing the testcases. In order to simplify this procedure, the PUV directory needs to be structured as shown in Fig. 3.

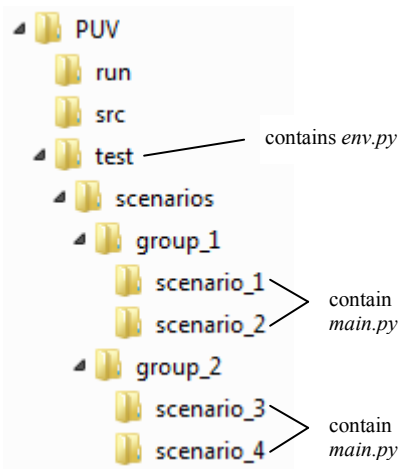

Figure 3: PUV File Hierarchy.

The *plcsim* script is started in the *run*\ directory. The *test*\ directory contains the environment in file *env.py*, and the *scenarios*\ directory contains the testcases, organized in groups. The testcase itself is defined in a file *main.py*, and the testcase name is deduced from the path to that file, for example "group_1\scenario_1". If *plcsim* is called without any arguments, then all testcases that are found below the *scenarios*\ directory are run. If an argument is specified, then only those testcases are run, whose name matches the argument. For example '*plcsim group_1*' would only run the testcases from test group *group_1*.

Once all files are located, a test suite is created and run with the list of the environment and testcase files. The suite executes all testcases sequentially, counts the passed and failed testcases and generates a final report, which is printed to the console.

Execution Environment

For PLC program development and verification a regular office PC (Intel Core i5 CPU, 8GB RAM) with Windows 7 64-bit OS is used. The PLC subsystem runs on a virtual machine with a Windows XP 32-bit OS, since TwinCAT 2.11 does not support x64 architecture. A future upgrade to TwinCAT 3.1 would eliminate the need for a separate VM, since it runs on x64 architecture.

In order to have support for the Python scripting language and tight integration into the .NET framework, IronPython [1] has been installed. The Beckhoff ADS .NET library is then simply referenced, and ADS methods can be called directly from Python code. Since the development of the verification environment is done with Visual Studio 2010, Python Tools for Visual Studio 2010 (PTVS) [2] has also been installed.

EXAMPLE

The PLC verification environment is used in the context of the European XFEL undulator control system. As described in the Conceptual Design Report [3], two PLC programs are used: one on local level to drive a single undulator, and one on global level to drive a complete undulator system. This example shows how verification has been set up for the global PLC program.

Setting Up Verification

The first step is to create the verification environment, stored in *test\env.py*:

```
from pvl.core.testing import TestEnvBase
from pvl.vc.undulator import SoftwareDriver
from pvl.vc.system import PlcDriver

class Env(TestEnvBase):

    def __init__(self):
        super(Env, self).__init__()
        self.ams_net_id = "10.0.254.2.1.1"
        self.plc_driver = PlcDriver('generic PLC VC',
            self)
        self.sw_driver = SoftwareDriver(
            'SW Interface VC', self)

    def setup(self):
        super(Env, self).setup()
        self.plc_driver.reset_rt()
        self.plc_driver.start_rt()

    def teardown(self):
        super(Env, self).teardown()
```

The environment is defined as a class named *Env* derived from the *TestEnvBase* class. *__init__()* is the constructor, which contains the instantiation of the verification components. In this example, a generic PLC VC and a software interface VC is used. The arguments to the VCs are a descriptive name and the reference to the

ISBN 978-3-95450-139-7

environment using *self*. The *setup()* method is used to reset and start the PLC before a testcase to have a defined PLC state. Both, *setup()* and *teardown()*, are required to first call the implementation of the base class.

Next step is the definition of testcases. The very simple example shown here verifies, whether the program version can be read out from an internal variable. The testcase is defined in file *scenarios\ads\version\main.py*:

```
from pvl.core.testing import TestBase

class Test(TestBase):

    def run(self, env):
        """Verify that version string is non-empty."""
        ver = env.sw_driver.read_version()
        self.assertTrue(len(ver) > 0, 'zero length version')
```

A testcase is defined as a class named *Test*, derived from *TestBase*. The scenario is implemented in the *run()* method, which takes the environment as an argument. Inside the scenario, the VCs can be accessed for example as *env.sw_driver*.

Each VC provides high level methods to interact with the PUV. In this case, the *read_version()* method of the SW interface VC is used to read the version string from the PUV. The name of the variable containing the version string is defined inside the VC. If this name is changed, then only the VC requires an update.

Subsequently, an assertion is used to verify, that the length of the version string is larger than zero. Assertion methods are defined in the *TestBase* class and can be called using the *self* reference.

Running Testcases

Testcases are run by changing to the *run* directory an executing the *plcsim* script:

```
C:\ >cd run
C:\ >plcsim ads\version
plcsim v0.1 (c) 2013 A.Beckmann
connecting to AMS net id: 10.0.254.2.1.1
Test: ads\version: FAIL -- 'Ads-Error 0x710 : Symbol could not be found.'
Group: ads: Run: 1 Failed: 1

Suite: 0.0% (0/1 passed)
  ads: 0.0% (0/1 passed)
```

In this case, only one testcase is selected by passing the name of the testcase as an argument to *plcsim*. *plcsim* command first connects to the PLC subsystem. Then, the testcase is run and the result is reported by a line in the format '*Test: <name of testcase>: <result>*'. Here, the testcase failed, because the variable defining the PLC program version string was not found.

After fixing the problem in the PLC program and rerunning the testcase, the output looks like:

```
C:\ >plcsim ads\version
plcsim v0.1 (c) 2013 A.Beckmann
connecting to AMS net id: 10.0.254.2.1.1
Test: ads\version: OK
Group: ads: Run: 1 Failed: 0

Suite: 100.0% (1/1 passed)
  ads: 100.0% (1/1 passed)
```

Now, the testcase passed and the version string functionality has been successfully verified.

OUTLOOK

The development of the automated PLC verification environment has just started. The architecture has been defined and an initial version of the PLC Verification Library has been created. Work will continue in order to improve usability, and also to make it independent from a specific PLC vendor.

One major enhancement would be to add a graphical user interface in order to improve the visualization of test results. Alternatively, it could be integrated into the Visual Studio IDE, which is used for TwinCAT 3.x, or any other IDE such as Eclipse or Netbeans.

Additional tools to automate the generation of verification environments, for example generating PLC subsystem configurations automatically from PUV code, would be of great help.

SUMMARY

Verification of PLC programs is an essential step in the development process of control systems. In order to reduce the effort to verify the functionality of the programs used in the European XFEL undulator control system, an automated verification environment has been set up.

The creation of the verification environment is facilitated by making common structures available through PVL, the PLC Verification Library. Functionality is verified by defining a set of testcases, which are run automatically using a single script. Testcases are self-checking, so that no manual work is required afterwards to decide, whether testcases have passed or failed.

Testcases run on simulated hardware, so that verification can be performed without the need for real hardware device.

REFERENCES

[1] http://www.ironpython.net
[2] http://pytools.codeplex.com
[3] A. Beckmann, S. Karabekyan, and J. Pflüger, „Conceptual Design Report Undulator Control Systems“, XFEL report XFEL.EU TR-2013-001, 2013

ISBN 978-3-95450-139-7