Whitepaper: Protocol-Based Verification for DCP

Martin Krammer, Clemens Schiffer
{martin.krammer,clemens.schiffer}@v2c2.at
Virtual Vehicle Research GmbH, Graz, Austria

March 30, 2021, Graz

Contents

1 Introduction

The *Distributed Co-Simulation Protocol* DCP is a platform and communication medium independent standard for the integration of models or real-time systems into simulation environments. It was specified in course of the ITEA 3 project ACOSAR. The DCP is standardized by the Modelica Association[1], where it is maintained as a Modelica Association Project[2] (MAP). It is designed to integrate models or real-time systems into simulation environments. It enables exchange of simulation related configuration information and data by use of an underlying transport protocol (such as UDP, TCP, or CAN). At the same time, the DCP supports the integration of tools and real-time systems from different vendors. The DCP is intended to make simulation based workflows more efficient, and to reduce the integration effort.

More detailed information about the DCP can be found in [1, 3] and on the DCP website.

2 Target Audience

This document is targeted to anyone who aims at implementing the DCP specification. The target audience includes, but is not limited to:

- Software tool vendors
- Test system suppliers
- System integrators
- Simulation engineers
- Professionals and students

3 Software Packages and Relationships

3.1 Overview

Right from the beginning, the DCP was intended as an open-access specification, accompanied by open-source software. The following software projects emerged out of ITEA 3 ACOSAR, and are currently being maintained by Modelica Association.

- DCPLib[3] is an open-source software library written in C++. It provides the necessary packages to create both, DCP master and slave software components to realize distributed co-simulation.

[1]http://www.modelica.org
[2]http://www.dcp-standard.org
[3]https://github.com/modelica/DCPLib

- DCPTestGenerator[4] is an open-source software library written in Java. It transforms a DCP slave description (DCPX) file, a test procedure template, and a test procedure extension to an XML based test procedure. It consists of a graph structure, intended for slave verification.
- DCPTester[5] is an open-source software library written in C++. It is intended to consume the generated test procedure of the DCPTestGenerator, mount a DCP slave-under-test, and execute the test procedure.

3.2 Relationships

The basic idea of protocol-based verification for distributed co-simulation is explained in [2]. The relationship between the DCPTestGenerator and the DCPTester is shown in Figure 1.

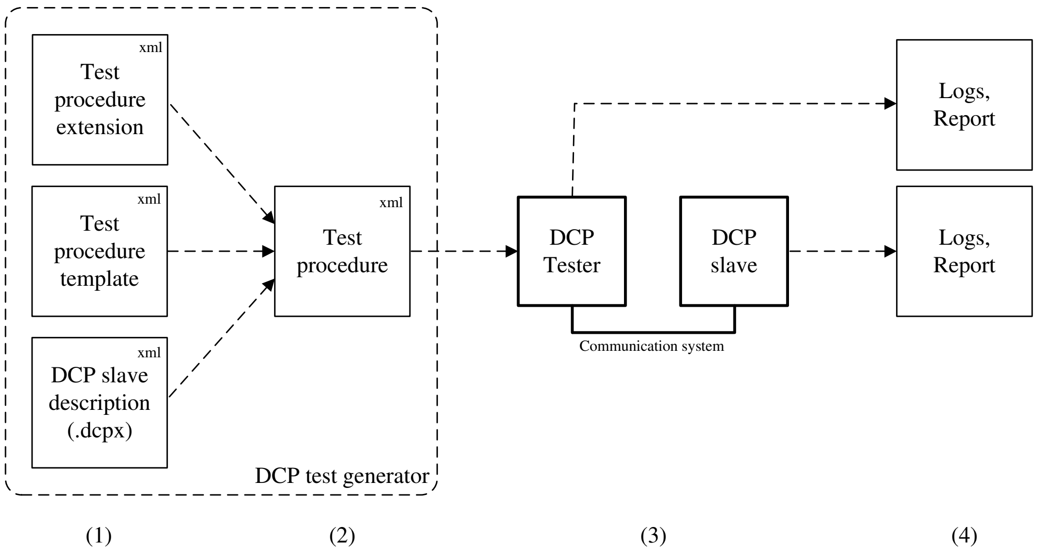

Figure 1: Relationship between DCPTestGenerator and DCPTester [2].

3.3 Target Version Information

This document matches the software release numbers provided on GitHub as follows (see Table 3.3).

[4]https://github.com/modelica/DCPTestGenerator
[5]https://github.com/modelica/DCPTester

Software	Version
DCPLib	v0.2
DcpTester	v0.2
DcpTestGenerator	v0.2

Table 1: Software target version numbers.

4 Scope and Non-Scope

This document is intended as an aid to set up protocol-based testing using the DCP. Its scope is defined in Figure 2. The dashed line shows the positioning of protocol-based verification in the complete V-diagram.

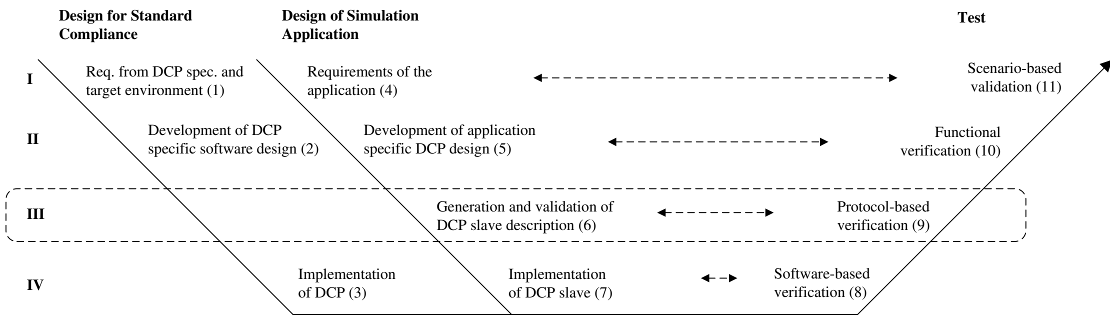
Design for Standard Compliance
Design of Simulation Application
Test
I
II
III
IV
Req. from DCP spec. and target environment (1)
Requirements of the application (4)
Scenario-based validation (11)
Development of DCP specific software design (2)
Development of application specific DCP design (5)
Functional verification (10)
Generation and validation of DCP slave description (6)
Protocol-based verification (9)
Implementation of DCP (3)
Implementation of DCP slave (7)
Software-based verification (8)

Figure 2: V-model for development, verification and validation of a DCP slave [2].

Its **non-scope** includes:

- Implementation of a slave and associated software-based verification
- Application and/or DCP-specific software design and associated functional verification
- Scenario-based validation

5 DCP Slave-Under-Test

The repository `DcpTester` provides a DCP slave for demonstration purposes. It is intended as a sample slave-under-test. Its DCP slave description (DCPX) file is also provided. It has the following features:

- Support for SRT and NRT operating modes
- Support for different resolutions (5/100s, 1/10s, 1/1000s)
- Support for transport protocol UDP IPv4 (TCP also possible)

- 12 Input Variables (one for each data type)
- 12 Output Variables (one for each data type)
- 10 Parameter Variables (one for each data type, excluding binary and string data types)
- A basic log category and template
- Accepts PDUs from Configuration PDU family (flag set)
- Can handle reset (flag set)
- Can handle variable steps (flag set)
- Can provide log on request (LoR, flag set)
- Can provide log on notification (LoN, flag set)

The provided DCP slave has no dedicated functionality and assigns the value of an input to the value of an output of the same data type.

6 Files and Structures

6.1 Test Procedure Template

6.1.1 Format

The *Test Procedure Template* can be used for creation of *Test Procedures.* In fact, it represents a minimum *Test Procedure*, as soon as a UUID of a slave-under-test is added to it, next to connectivity information (Element `TransportProtocols`)

Definition. The Test Procedure Template is defined by `DcpTestProcedure.xsd`

Definition. *Test Procedure Templates* file names shall comply to the following scheme: `tpl_*.xml`

6.1.2 Defined Test Procedure Templates

The test procedure templates provided are shown in Table 2. These templates are intended for extension by *Test Procedure Extensions.* This leads to the ability to test DCP slave behaviour in specific states.

Transitions to states that are left via self-triggered transitions (e.g. 4, `CONFIGURING`) can also be tested, but the slave must remain in this state. This could require a modification of the implemented slave (e.g. by adding delay), compared to a production slave.

File name	Short Description
tpl_min.xml	Minimal pass through state machine, no initialization
tpl_stop.xml	Multiple passes through state machine, stop in every state
tpl_to00alive.xml	Go to state 0 and stop
tpl_to01configuration.xml	Go to state 1 and stop
tpl_to03prepared.xml	Go to state 3 and stop
tpl_to05configured.xml	Go to state 5 and stop
tpl_to07Initialized.xml	Go to state 7 and stop
tpl_to10Synchronized.xml	Go to state 10 and stop
tpl_to16configured.xml	Go to state 16 and stop
tpl_to11configured.xml	Go to state 11 and stop, for NRT
tpl_to13configured.xml	Go to state 13 and stop, for NRT

Table 2: Defined *Test Procedure Templates*

6.2 Test Procedure Extension

6.2.1 Format

Definition. The *Test Procedure Extensions* are defined by `DcpTestProcedureExtension.xsd`

Definition. *Test Procedure Extensions* file names shall comply to the following scheme: `ext_*.xml`

6.2.2 Defined Test Procedure Extensions

List of available extensions

File name	Short Description
ext_Min.xml	Add minimal slave specific data
ext_ValidConfiguration.xml	Provides a minimal valid configuration (not required when `canAcceptConfigPdus` set to false)
ext_negativeAllowedPDUsAlive.xml	Test rejection of not allowed PDUs in state 0
ext_negativeAllowedPDUs01Configuration.xml	Test rejection of not allowed PDUs in state 1
ext_negativeAllowedPDUs03Prepared.xml	Test rejection of not allowed PDUs in state 3
ext_negativeAllowedPDUs05Configured.xml	Test rejection of not allowed PDUs in state 5
ext_negativeInvalidConfiguration.xml	Test invalid configuration data
ext_negativeInvalidValues00Alive.xml	Test invalid values in state 0
ext_negativeInvalidValues01Configuration.xml	Test invalid values in state 1
ext_negativeInvalidValues03Prepared.xml	Test invalid values in allowed PDUs
ext_negativeInvalidValues05Configured.xml	Test invalid values in allowed PDUs
ext_negativeInvalidValues07Initialized.xml	Test invalid values in allowed PDUs
ext_negativeInvalidValues10Synchronized.xml	Test invalid values in allowed PDUs
ext_positiveInit.xml	Test initialization loop (100 iterations)
ext_stop.xml	Stop in every state (intended to be used with `tpl_stop.xml`)

Table 3: Defined *Test Procedure Extensions*

6.3 Test Procedure

6.3.1 Format

If slave specific information is added to the test procedure template, it results in a slave-specific test procedure.

Definition. The Test Procedure is formally defined by `DcpTestProcedure.xsd` (same schema file as in 6.1).

Definition. *Test Procedure* file names shall comply to the following scheme: `proc_*.xml`

6.4 Exhaustive Testing

The *DCP Test Generator* was initially released with a set of default templates and extensions. They were meant for exhaustive testing, as described in [2]. The delivered files are shown in Table 4.

File	Description
`predefined_template_withReset.xml`	Covers one pass through the DCP state machine, including the reset transition.
`predefined_template_withoutReset.xml`	Covers one pass through the DCP state machine, excluding the reset transition.
`predefined_extension_SRT_TCP.xml`	Extends the templates by steps causing error codes, for TCP transport protocol.
`predefined_extension_SRT_UDP.xml`	Extends the templates by steps causing error codes, for UDP transport protocol.

Table 4: Templates and extensions for exhaustive test.

7 How to Create a Custom Template or Extension

7.1 Default Extension Mechanisms

In this section we want to show an example, how to create a custom template, an extension, and how this results in a new test procedure. The goal is to send the slave a PDU in state `CONFIGURATION` with invalid content and check if the slave reacts with the corresponding error code. Listing 1 shows an excerpt of a test procedure template. The template represents the registration procedure of DCP. After registration the slave transitions to `CONFIGURATION` state. In order to do so, the values of `STC_register` are generic and need to be replaced with slave specific values. The template expects to exit the test procedure at step 49 (see at the end of line 2).

Listing 1: Snippet of a template

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <DcpTestProcedure version="0.0" name="allPaths.xml"
     acceptingSteps="49">
3     <TransportProtocols/>
```

```
    <Transition from=”0” to=”1”>
        <Sending>
            <STC_register receiver=”1” state_id=”0” op_mode=”0”
                 major_version=”0” minor_version=”0”/>
        </Sending>
    </Transition>
    <Transition from=”1” to=”2”>
        <Receiving>
            <RSP_ack sender=”1”/>
        </Receiving>
    </Transition>
    <Transition from=”2” to=”4”>
        <Receiving>
            <NTF_state_changed sender=”1” state_id=”1”/>
        </Receiving>
    </Transition>
    ...
```

Listing 2 shows an test procedure extension for the template shown before. In lines 1 − 14 slave specific data from the slave description is used to supplement the template in order to produce a well formed `STC_register`-PDU. This is needed as attributes like, e.g., the `UUID` represents a slave-specific information, that is not part of the template. Any transition of the template, which fulfills the given condition, will be updated with corresponding values of the slave description. The parameter `state` of `ExtensionSet` set to 1 extends all `CONFIGURATION` states. It adds several new transitions, with the goal to provoke different error codes in state `CONFIGURATION`. The example shows a situation where the tester sends a PDU (`STC_prepare`) with a state id of 0. Since the slave is in state 1 (`CONFIGURATION`), it is specified to reject it with the corresponding error code 8205. At the end, `UpdateMaxStep` is set to 2, because this branch of the tree data structure is extended by 2 steps.

Listing 2: Snippet of an extension

```
1  <ForEach variableName=”transition”>
2    <Set>
3      <Test.transitions/>
4    </Set>
5    <If
6      condition=”transition.sending!=undefined&amp;&amp;
           transition.sending.stcRegister!=undefined”>
7      <Update oldVariable=”transition.sending.stcRegister.
           minorVersion”
8          newVariable=”slave.dcpMinorVersion”/>
9      <Update oldVariable=”transition.sending.stcRegister.
```

```
            majorVersion”
10          newVariable=”slave.dcpMajorVersion”/>
11        <Update oldVariable=”transition.sending.stcRegister.
            slaveUuid” newVariable=”slave.uuid”/>
12        <Update oldVariable=”transition.sending.stcRegister.
            opMode” newVariable=”1”/>
13      </If>
14    </ForEach>
15
16    <ExtensionSet state=”1”>
17      <AddTransition fromOffset=”0” toOffset=”1”>
18        <Sending>
19          <STC_prepare>
20            <Receiver>
21              <Value value=”1”/>
22            </Receiver>
23            <StateId>
24              <Value value=”0”/>
25            </StateId>
26          </STC_prepare>
27        </Sending>
28      </AddTransition>
29      <AddTransition fromOffset=”1” toOffset=”2”>
30        <Receiving>
31          <RSP_nack>
32            <Sender>
33              <Value value=”1”/>
34            </Sender>
35            <ErrorCode>
36              <Value value=”8205”/>   <!-- INVALID_STATE_ID -->
37            </ErrorCode>
38          </RSP_nack>
39        </Receiving>
40      </AddTransition>
41      <UpdateMaxStep increase=”2”/>
42      ...
```

Listing 3 shows the resulting test procedure. Note the filled in slave specific data in `STC_register` in line 3. This is followed by permissible state transitions. In lines $15-19$, however, a `STC_prepare` is sent to the slave-under-test, requesting a correct state, but stating an incorrect current state in the attribute `state_id`. If responding correctly, the slave-under-test should reply using error code 8205.

Listing 3: Snippet of the finished procedure

```
    <Transition from="0" to="7">
        <Sending>
            <STC_register receiver="1" state_id="0" slave_uuid=
                "0d7217ea-ac72-11ea-bb37-0242ac130002" op_mode="
                1" major_version="1" minor_version="0"/>
        </Sending>
    </Transition>
    <Transition from="7" to="8">
        <Receiving>
            <RSP_ack sender="1"/>
        </Receiving>
    </Transition>
    <Transition from="8" to="36">
        <Receiving>
            <NTF_state_changed sender="1" state_id="1"/>
        </Receiving>
    <Transition from="36" to="58" log="false">
        <Sending>
            <STC_prepare receiver="1" state_id="0" />
        </Sending>
    </Transition>
    <Transition from="58" to="59" log="false">
        <Receiving>
            <RSP_nack sender="1" error_code="8205"/>
        </Receiving>
    </Transition>
    ...
```

7.2 Advanced Extension Mechanisms

In Listing 4 an alternative extension is shown demonstrating the loop capabilities of extensions. Rather then checking for one specific value of `state_id` (as shown in Section 7.1, it is possible to check for a set of incorrect values. The `ForEach` statement iterates over all possible DCP states (0 − 18, line 4), except for the current state (line 6). The slave is expected to provide the correct error code (8205) for all of these cases.

With this tool many transitions can be generated at once allowing for effective testing of the protocol.

Listing 4: Snipped of extension showing loops.

```
1  <ExtensionSet state="1">
2    <ForEach variableName="oState">
3      <Set>
```

```
        <DcpStates/>
      </Set>
     <If condition="state!=oState">
      <AddTransition fromOffset="0" toOffset="1">
        <Sending>
          <STC_prepare>
            <Receiver>
              <Value value="1"/>
            </Receiver>
            <StateId>
              <Variable variablenname="oState"/>
            </StateId>
          </STC_prepare>
        </Sending>
      </AddTransition>
      <AddTransition fromOffset="1" toOffset="2">
        <Receiving>
          <RSP_nack>
            <Sender>
              <Value value="1"/>
            </Sender>
            <ErrorCode>
              <Value value="8205"/> <!-- INVALID_STATE_ID -->
            </ErrorCode>
          </RSP_nack>
        </Receiving>
      </AddTransition>
      <UpdateMaxStep increase="2"/>
    </If>
    </ForEach>
```

It is advantageous to use such a dynamic generation of test procedures, especially when testing the configuration of slaves, e.g., iterating over the set of all inputs, and all possible configurations for these inputs. Possible sets to iterate are given in Table 7.2.

Iterable set
Slave.Outputs
Slave.Inputs
Slave.Parameters
DcpStates
DcpOpModes
DcpTransportProtocols
Test.transitions

Table 5: Iterable sets for extensions

Another mechanism is the definition of ranges, so it is possible to iterate over a range of integers, as shown in Listing 5. Using this technique, we can test the entire remaining range of invalid state identifiers, starting from 19, up to the highest possible (8-bit integer) number 255.

Listing 5: Snipped of extension showing a loop over a range

```
  ...
 <ForEach variableName="invalidState">
 <Set>
     <IntegerRange min="19" max="255"/>
 </Set>
 <AddTransition fromOffset="0" toOffset="1">
   <Sending>
     <STC_prepare>
       <Receiver>
         <Value value="1"/>
       </Receiver>
       <StateId>
         <Variable variablenname="invalidState"/>
       </StateId>
     </STC_prepare>
   </Sending>
 </AddTransition>
  ...
```

8 Docker Container-based Testing

8.1 Basic Procedure

The docker container builds the *DCP Library*, the *DCP Tester*, the *DCP Test Generator* and the provided slave-under-test. The usage of a docker container has two advantages:

1. It provides a platform independent encapsulated environment, in which all software components can be build, under consideration of their dependencies.
2. The test procedures are executed on the slave-under-test.
3. At the same time, it provides documentation of the build process, as the Docker-file describes the environment.

The docker can be build as follows (see Listing 6).

Listing 6: Building the Docker container

```
git clone https://github.com/modelica/DCPTester.git
cd DCPTester/tester/docker
docker-compose build
```

Once the build is complete the docker can be started in interactive mode as shown in Listing 7.

Listing 7: Building the Docker container

```
docker-compose run Test bash
```

This results in the following folder hierarchy:

- `/cmake`
- `/src/dcptester`
- `/tester/Extensions`
- `/tester/Templates`
- `/tester/SlaveUnderTest`

The folders `Extensions`, `Templates`, and `Procedures` contain the files and structures explained in Section 6. The folder `Scripts` contains 2 shell scripts.

Listing 8: Test procedure generation

```
cd Scripts
./generate_Tests.sh
```

Line 2 of Listing 8 calls the DcpTestGenerator, consumes extensions, templates, and a DCPX file. The file `TestsToGenerate.csv` describes which templates are to be combined with which extensions. The script iterates over this list and produces a set of test procedures, and writes them to a file (`ListOfTests.txt`).

After generation, the following directories are generated:

- `/tester/Scripts/Procedures`

- /docker/Shared

Listing 9: Test procedure generation

```
./run_list_of_tests.sh
```

Listing 9 executes the slave-under-test and the DcpTester, which consumes the previously generated test procedures. It produces a set of log files in the folder `logs`. After script execution, this folder will hold log files for each test procedure, collected from both the slave-under-test and the tester. The file names from the tester follow the format `tester_proc_ext_*.log`. The file names from the slave-under-test follow the format `slave_proc_ext_*.log`. The file `summary.log` provides an overview about all executed test procedures. It provides information about the executed test procedures.

Alternatively, all these steps are carried out by issuing the command shown in Listing 10 on the Docker host system to start the service.

Listing 10: Start Docker container and execute the defined service.

```
cd DCPTester/tester/docker
docker-compose up
```

8.2 Important Notice on Test Results

In file `summary.log` the outcome of each test procedure is shown. The possible outcomes are as follows.

`SUCCESS` The test procedure was executed and execution finished in one of the defined accepting steps.

`REVIEW` The test procedure was executed and execution did not end in a defined accepting step. This can have two causes:

- The tester experienced a step that is not defined in the procedure. It is important to note that the slave behaviour may still be compliant to the DCP specification. Review of the log files is highly recommended. This situation may occur due to non-deterministic behaviour of, e.g., operating system schedulers, network stack implementations, or other sources.
- The tester experienced a step that is not defined in the procedure and violates the DCP specification. Review of the log files is highly recommended, to determine if modifications of the slave-under-test are needed.

References

[1] Martin Krammer, Martin Benedikt, Torsten Blochwitz, Khaled Alekeish, Nicolas Amringer, Christian Kater, Stefan Materne, Roberto Ruvalcaba, Klaus Schuch, Josef Zehetner, Micha Damm-Norwig, Viktor Schreiber, Natarajan Nagarajan, Isidro Corral, Tommy Sparber, Serge Klein, and Jakob Andert. The Distributed Co-simulation Protocol for the integration of real-time systems and simulation environments, 2018.

[2] Martin Krammer, Christian Kater, Clemens Schiffer, and Martin Benedikt. A Protocol-Based Verification Approach for Standard-Compliant Distributed Co-Simulation. In *Proceedings of Asian Modelica Conference 2020, Tokyo, Japan, October 08-09, 2020*, volume 174, pages 133–142, 2020.

[3] Martin Krammer, Klaus Schuch, Christian Kater, Khaled Alekeish, Torsten Blochwitz, Stefan Materne, Andreas Soppa, and Martin Benedikt. Standardized Integration of Real-Time and Non-Real-Time Systems: The Distributed Co-Simulation Protocol. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4-6, 2019*, volume 157, pages 87–96. Linköping University Electronic Press, Linköpings universitet, feb 2019.