A Protocol-Based Verification Approach for Standard-Compliant Distributed Co-Simulation

Martin Krammer[1] Christian Kater[2] Clemens Schiffer[1] Martin Benedikt[1]

[1]Virtual Vehicle Research GmbH, Co-Simulation and Software Group, Austria,
{martin.krammer,clemens.schiffer,martin.benedikt}@v2c2.at
[2]Comsysto Reply GmbH, Germany, c.kater@reply.de

Abstract

The Distributed Co-Simulation Protocol (DCP) is a platform and communication system independent application level communication protocol. It is designed to integrate models or real-time systems into simulation environments. The specification document defines the structure and behaviour of slaves. It is advantageous to verify and validate a slave before integrating it into a larger co-simulation scenario. This is especially true if tests are performed on large rigs, where availability and time are typically limited. Until now, no systematic procedure for design, verification and validation of slaves is available.

In this paper, we introduce a process for design, verification and validation of DCP slaves. The process is used to systematically encapsulate models or real-time systems into slaves. For verification, the DCP state machine and protocol definitions are used to derive sequences of protocol data units (PDU) from any given DCP slave description. To demonstrate the feasibility of our approach, we show a use case from the automotive engineering domain. It includes a slave representing an engine, which is embedded using the Functional Mock-Up Interface (FMI). We also introduce the *DCP test generator* and *DCP tester* tools to automate the verification steps. With these contributions, slaves can be developed systematically and more efficiently. The introduced software is available under an open-source license.

Keywords: co-simulation, simulation, test, real-time, distributed

1 Introduction

Co-simulation-based methodologies have evolved significantly during the last decade. Nowadays co-simulation is a major enabler for holistic cross-domain or system simulations. It allows integration of simulation models, tools, and solvers from different sources.

The functional mock-up interface (FMI) (Blochwitz et al., 2011) represents an important software standard for co-simulation in several industry sectors. It was proposed to solve the need for interoperability between models, solvers and tools. FMI was developed in the MODELISAR project, starting in 2008. The FMI specification is standardized as a Modelica Association Project (MAP). Its most recent specification version is 2.0.1 which was released in 2019. The FMI specification document defines an interface for model exchange and co-simulation. Today more than 100 software tools support the FMI[1]. For distributed simulation environments, network communication technologies are frequently used in practice. However, "the definition of this communication layer is not part of the FMI standard" (Modelisar Consortium and Modelica Association Project "FMI", 2019, p.95).

The Distributed Co-Simulation Protocol (DCP) fills this technology gap. It was developed in the ACOSAR project (Krammer et al., 2016). The DCP is an application-level communication protocol 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 reduce the overall system integration effort. It was designed with FMI compatibility in mind, i.e., it follows a master-slave communication principle, uses an aligned state machine implementing an initialization mechanism, and defines an overall integration process which is driven by standardized XML file formats. Version 1.0 of the DCP specification document was released as an open-access Modelica standard in early 2019 (Krammer et al., 2018a, 2019).

2 Motivation

The DCP specification document describes the design of a slave only. A master is required to control a co-simulation scenario, which includes at least one slave. A slave encapsulates a model or real-time system. It therefore represents a simulation subsystem providing standardized access capabilities. The subsequent paragraphs motivate testing for slaves.

[1]http://fmi-standard.org/tools/

Development Support The initial need to automate protocol-based verification emerged out of the ACOSAR project. The specification document was engineered using a requirements-based approach (Krammer et al., 2018b). During development of the DCP new sets of technical requirements led to new specifications, which in turn raised demand for testing. First approaches included a tester tool able to send and receive predefined sequences of protocol data units (PDU) to and from a single slave. This fulfilled initial requirements for simple and configurable tests. Nevertheless that approach still demanded manual development of test cases, which caused high effort while suffering from low test coverage.

Protocol Implementation For full standard compatibility, DCP implementations must behave as specified and meet the standard requirements. The DCP is a platform and programming language independent specification. So it makes sense to test implementations at protocol level.

Application Specific Adaptation If a DCP slave encapsulates a real-time system, functional integrity is very important. On one hand, a DCP slave needs to behave as intended, e.g., connect to specific variables at the given communication step size. On the other hand, missing or delayed data packets, broken physical connections, etc. need to be handled properly if no other mechanisms are in place to avoid damage to equipment and operators. To handle these issues, the DCP features several mechanisms, like separate states for error handling and recovery. However, the criteria for transitioning to these states are application dependent and must be developed accordingly. Furthermore, all possible real-time system hazards must be analyzed, and potential safety measures implemented. Therefore, adaptations and safety measures are subject to test, in order to assure their functionality during operation.

Limited Access Access to industrial models or real-time systems might be limited, due to high capacity utilization or high rental cost. Typical examples include physical appliances like various kinds of automotive test beds, roller rigs, brake dynamometers, or driving simulators, to name a few. If DCP slaves encapsulate such systems, or are used in connection with such systems, it seems reasonable to test the DCP slave prior to scenario integration, in order to save time and money.

Economy For aforementioned reasons it seems reasonable to test DCP slaves with the goal to fix as many defects as early as possible. Delivering a reliable DCP slave is not only beneficial for the integrator, but also for the provider. From an economic perspective the overall number of development cycles is reduced and time-to-market is improved.

This paper contributes in two ways. Our goal is to describe a systematic development process for DCP slaves and how created DCP slaves can be tested prior to scenario integration. Therefore we introduce (1) a generic DCP slave development process and (2) a methodology and two coordinated tools for test of DCP slaves. Section 3 recapitulates related work. Section 4 describes the process main phases. Section 5 defines the main concepts of protocol based verification. Section 6 introduces the DCP Test Generator and Tester tools. Section 7 describes a use case from the automotive domain. Finally, Section 8 concludes this paper.

3 Related Work

A survey of communication protocol testing is presented in (Lai, 2002). It focuses on test sequence generation methods, test coverage, fault models and prediction, test tools, and experience reports. (Bochmann and Petrenko, 1994) provide a good overview of methods and their relevance for software testing. (Sidhu and Leung, 1989) give four formal methods for protocol testing by developing test sequences. (Linn, 1989) introduces a conformance evaluation methodology for protocol testing.

In the context of FMI, many efforts for testing functional mock-up units (FMU) have been made. The FMI Cross-Check repository[2] contains a large number of test FMUs provided by different vendors. The repository holds exported FMUs and results for imported FMUs of the tools that take part in this initiative. Therefore the contained FMUs can be used to test and improve the interoperability of FMI compatible import- and export-capable tools. The FMI Compliance Checker[3] is intended for FMU validation. It checks for FMI 1.0 and 2.0 compliance to the standard specification. Its basic features include XML model description checking and validation of binary FMUs. For the latter, it is able to load a binary module and check for the availability of all required functions. For an FMU for model exchange, the checker tool tests for explicit Euler numeric integration capability. For an FMU for co-simulation, the checker tool tests for fixed step-size calculation capability. Furthermore, the checker tool is able to provide numerical input data and log computed solution outputs. FMPy[4] is a Python library by Dassault Systèmes to validate and simulate FMUs. It has a graphical user interface, compiles C code FMUs and generates CMake projects for debugging.

To our best knowledge, a dedicated DCP test approach or tool is not available at this point in time.

[2] https://github.com/modelica/fmi-cross-check

[3] https://github.com/modelica-tools/FMUComplianceChecker

[4] https://github.com/CATIA-Systems/FMPy

DOI 10.3384/ecp20174133

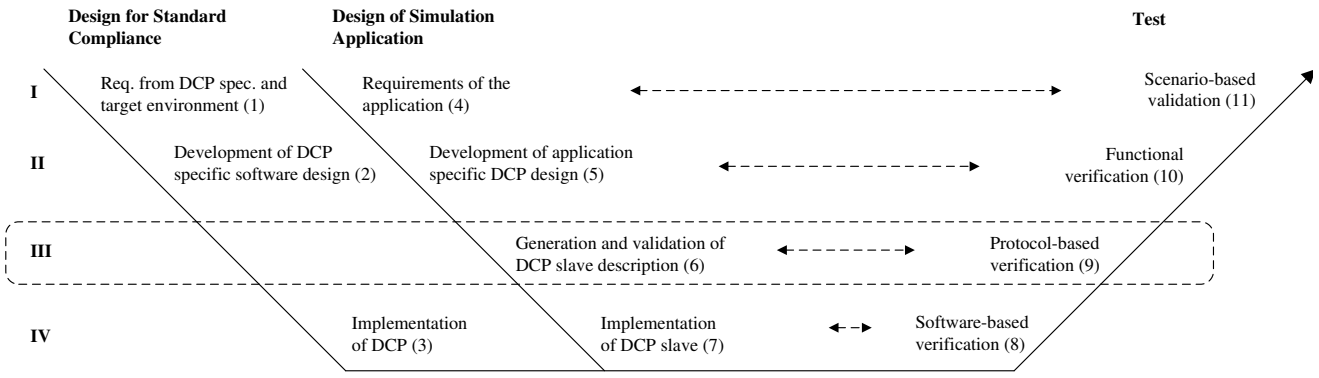

Figure 1. V-model for development, verification and validation of a DCP slave.

4 DCP Slave Development Process

4.1 Overview

We propose a V-model for development, verification and validation of DCP slaves. The classical V-model describes activities in development projects. It is traversed over time. Activities on the left-hand side are related to specification and design, whereas activities on the right-hand side are related to test. Each specification activity corresponds to one test activity. Figure 1 shows a V-model for design and test of a DCP slave. The V-model has four levels, ranging from requirements engineering and scenario (I), development and functions (II), data model and protocol (III), to implementation and software (IV). Its left side is split into two branches. Activities (1)-(3) are related to *Design for Standard Compliance*. They are used to implement the DCP according to the specification document, without any concrete simulation application. Activities (4)-(7) are related to *Design of Simulation Application*. They are used to tailor an existing DCP implementation according to the required simulation application. Its right side is dedicated to *Test*. Activities (8)-(11) are performed subsequently after implementation, to verify and validate the resulting slave.

Due to the structure of the V-model, it can be tailored to various needs. It can be used to implement the protocol first, and add the application specific parts afterward. The DCPLib represents an example of this approach. Alternatively, it might be desirable to have a monolithic implementation, where the simulation application directly implements standard-compliant behaviour. Examples include electronic control units (ECU) or microcontrollers providing DCP access. Subsequently, each of the V-model activities are described in detail.

4.2 Design for Standard Compliance

4.2.1 Requirements from DCP Specification and Target Environment

The DCP standard consists of a data model, a finite state machine, and a communication protocol including a set of protocol data units. The specification document defines how these parts interact with each other. These requirements are independent from an actual simulation application, and are mandatory for implementation. Additionally, requirements of the target environment will affect subsequent development. This includes e.g., the target operating system, preferred programming languages, available software modules or libraries, communication systems, and transport protocols.

4.2.2 Development of DCP Specific Software Design

This activity should clarify how to implement the DCP. The main result is a software architecture. DCP PDUs may be implemented using object-oriented concepts of classes and inheritance, in contrast to using pre-defined arrays. For send and receive functionality, an external application programming interface might be used. Depending on the target platform, concurrency can be exploited. For example, threads can be used to send and receive data at regular intervals.

4.2.3 Implementation of DCP

During this activity the DCP is implemented. The resulting work product of this activity is a compiled library or software module. It is not able to run on its own, as simulation application-specific information is missing.

4.3 Design of Simulation Application

4.3.1 Requirements of the Simulation Application

In this activity, the requirements of the simulation application are determined. This includes the identification of the simulation model or real-time system

for encapsulation, and the identification of quantities for simulation data exchange.

4.3.2 Development of Application Specific DCP Design

In this activity, several slave internal properties are defined. First, this includes the definition of variables and their causality (inputs, outputs, parameters and tunable parameters), mapping of these variables to variables of the encapsulated model or real-time system, and variable properties, like step size in combination with the slave's specified time resolution. Second, the finite state machine must be adapted to the behaviour of the encapsulated model or real-time system. Third, according to the specification, each state of `Normal Operation` requires a well-defined transition to the `Error Handling` state, where appropriate actions for recovery must be defined. Finally, if the intended slave should support starting simulations from a non-trivial initial condition, then an initialization/synchronization strategy must be elaborated.

4.3.3 Generation and Validation of DCPX

The DCP slave description (DCPX) is an XML (Extensible Markup Language) file which describes one single DCP slave. It contains all static information related to one specific DCP slave. Its structure is defined by a normative XML XSD (XML Schema Definition) file (Krammer et al., 2019). It not only defines the required structures of elements and attributes, but also supplementary assertions and constraints. Assertions and constraints are well suited for expressing logical relationships between elements and attributes.

Assertions are expressed in the `xs:assert` tag using the XML Path Language (XPath). They are a feature of XSD version 1.1. Assertions constrain the existence and values of related elements and attributes. Furthermore, `xs:unique`, `xs:key` and `xs:keyref` tags are used to express constraints. Typical examples of application include the verification of uniqueness of names and the verification of cross-referenced key values.

In the context of the DCP specification, assertions and constraints provide strong formalisms which can be used for automated validation, whenever a DCPX file is generated or imported.

4.3.4 Implementation of DCP Slave

In this activity the slave is implemented. As a typical resulting work product, an executable is created out of production code. A DCP slave is defined as either a simulation model or a real-time system on a ready-to-run execution platform that is accessible via DCP over a supported communication medium. Therefore the slave software might be deployed on its target platform.

4.4 Test

4.4.1 Software-based Verification

This activity is dedicated to software verification approaches. Procedures and processes for software verification are covered in (Rakitin, 2001; Myers et al., 2011). For example, inspections can be used to monitor the design for standard compliance.

4.4.2 Protocol-based Verification

This activity is situated between software-based and function-based verification. Details are presented in Section 5.

4.4.3 Function-based Verification

If protocol-based verification is successfully passed, testing may continue with function-based verification. Function-based verification refers to the functionality of the connection to the underlying model or real-time system. For that purpose, a dedicated master shall be able to configure and stimulate a slave's inputs and parameters by sending PDUs. This master must also observe the slave's response by monitoring its outputs. If defined, all variables must be within their thresholds, like minimum or maximum values. Furthermore, available dependency information can be used for testing as well. The outputs of a DCP slave may depend on its inputs and parameters.

4.4.4 Scenario-based Validation

In this activity, the developed slave is instantiated and integrated into the overall co-simulation scenario. A co-simulation scenario is defined as the integration of multiple DCP slaves to perform a common simulation task. A capable master is required, to perform registration, configuration, initialization, synchronization and the actual simulation.

5 Protocol-based Verification

5.1 Finite State Machine Based Approach

The main idea of protocol-based verification is to check if a DCP slave behaves according to the standard's specification document. This includes the communication protocol, the state machine and its transitions, and the slave's configuration. To check a slave for protocol compliance, the concept of a tester was developed. The tester should connect to a slave and stimulate it by sending PDUs. However, sending plain, pre-defined sequences of PDUs to slaves for protocol-based verification is not reasonable. A DCP slave may expose non-deterministic behaviour, due to e.g., network delay, affecting the protocol and state machine (Modelica Association Project DCP, 2019, p. 21). Figure 2 shows an example of such a situation. After a state change request from state CONFIGURED to state INITIALIZING, a slave acknowledges the request and performs the transition (steps 16 to

DOI 10.3384/ecp20174133

19). In INITIALIZING it performs the necessary calculations, performs a self-triggered state transition to INITIALIZED and sends a notification including the new current state identifier (step 19 to step 20). Another possibility would be that the master decides to stop during the Initialization superstate. It sends STC_stop to the slave (step 19 to 21). The slave may acknowledge this request, and perform the state transition to STOPPING as requested (steps 21, 22, 23). However, if computation for initialization ends before STC_stop can be processed, the slave transitions to INITIALIZED and sends a corresponding notification. A negative acknowledgment including an error code follows (steps 21, 24, 25).

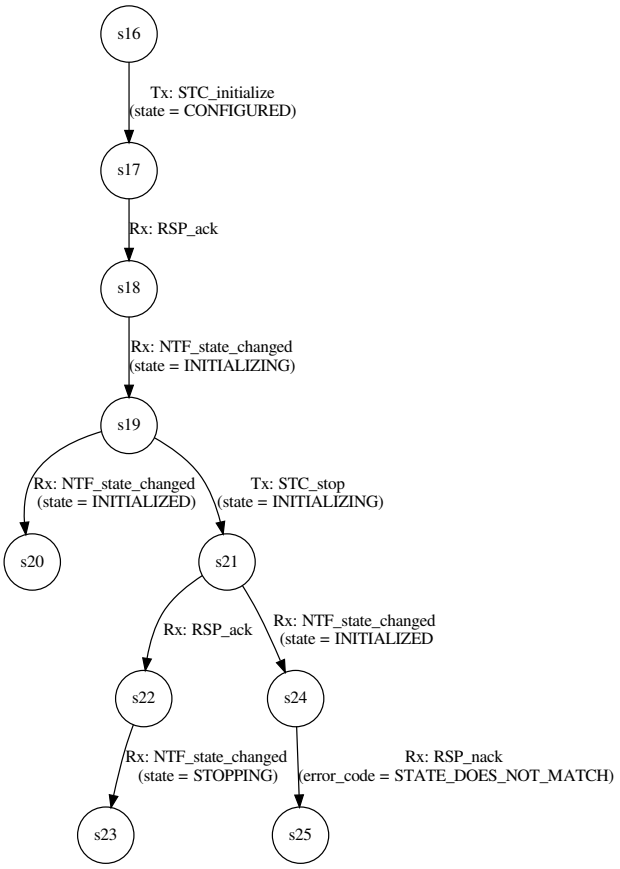

Figure 2. Example for multiple options to pass through the DCP state machine.

Protocol-based verification must consider such cases and react accordingly. That includes scalable time frames for request and response PDUs.

5.2 DCP Test Procedure

To circumvent the previously explained problem and to cover all possible situations, a DCP test procedure is defined. It is based on a finite automaton, which defines *steps* for protocol verification. Each step has a successor step. To proceed from step to step, transitions are defined. A transition can be triggered by either sending or receiving a protocol data unit (PDU). The DCP uses the concept of PDUs which are exchanged between DCP slaves and the DCP master. Since the DCP not only covers the exchange of simulation data (e.g., inputs/outputs, parameters), but also the set up (e.g., the configuration of inputs/outputs) and control (e.g., start and stop commands) of a co-simulation scenario, 34 different PDUs are organized in PDU families. A taxonomy is shown in (Krammer et al., 2018a). Certain PDU types are only meant to be sent one-way, from the sender to the receiver. For example, PDUs of the configuration family are only sent from the DCP master to the slave. Other PDU types may be sent in both directions. For example, PDUs of the data family may be used to exchange simulation data between slaves, but also between the master and a slave. The finite automaton of the DCP test procedure allows to define arbitrary PDU exchanges. Additionally it includes the concept of time to define a period after which a transition should be performed. Every finite automaton expressed using the DCP test procedure has defined steps for entry and exit. A valid step to exit the finite automaton is denoted here as an *accepting step*. A test procedure is successfully executed if the finite automaton is left through such an accepting step.

Formally, a DCP test procedure is defined by the following 6-tuple $(MaxStep, F, C, M, \delta, \lambda)$, where

- $MaxStep$ is the upper bound of steps in the procedure. The set of steps S is defined as

  $$S = \{n \mid n \in \mathbb{N}_0 \wedge n < MaxStep\}$$

  The initial step number equals to zero.

- F is the set of accepting steps. The test procedure is successful if it finalizes in an accepting step.

- C is a set of time points (unit: seconds) at which a transition from one step to another is performed.

- M is a set of PDUs according to the DCP specification which are sent or received within the test procedure.

- δ is a transition function between the steps of the test procedure:

  $$\begin{aligned}\delta &\subseteq S \times \Sigma \to S \\ \Sigma =&(\{Receive\} \times M) \cup \\ &(\{Send\} \times (C \cup \{-\}) \times M)\end{aligned}$$

- λ defines whether a transition should be considered in the statistical evaluation of the procedure:

  $$\lambda = S \times \Sigma \to \{true, false\}$$

5.3 Test Procedure Extension

Generating a generic test procedure is not feasible, because different slaves have different features which need to be considered. Furthermore, some parts of the test procedure will appear repeatedly. A typical example is to test if a slave correctly repudiates incorrect requests for state changes, in all of its states. Therefore extensions to DCP test procedures may be defined. An extension contains a description on how to extend a given test procedure, together with the DCP slave description of the slave under test. The main idea is to take a basic sequence through the state machine, modeled as a test procedure, and extend this sequence with slave specific aspects and repetitive parts. The main control elements to define an extension are:

- *ExtensionSet* contains a sequence of operations which has to be executed on every step which has transition with an `NTF_state_changed` PDU as predecessor, given by a certain state. The emerging transitions will be put in between the considered step and its successor. Algorithm 1 shows how this can be implemented in a generator for test extensions.
- *AddTransition* adds a transition between two steps. The transitions needs to be bound to a receiving or sending PDU, as well as the information that the occurrence of this transition shall be logged. The fields of the PDU can be specified by:
  - $Value$: a specific value for the field.
  - *Random*: a random, invalid value.
  - $Variable$: the value of the field is read from a variable.
  - *Invalid*: a invalid value for this field.
- *Update* overwrites the value of a given field of the PDU inside a transition. This can be used to update specific fields, which need to be valid in a transition. For example, the `STC_register` PDU is needed in a basic path through the state machine. But the UUID of the slave can not be known upfront. To solve this, *Update* can be used.
- *UpdateMaxStep* Increases the MaxStep attribute by a given value.
- If allows conditional execution of control elements.
- $ForEach$ loops over a given set of items, like the states of the DCP protocol.

In addition to these control elements a test procedure extension can also access the DCP slave description to use and check against specific properties of the tested slave.

5.4 Test Procedure Algorithm

An algorithm to operate on the formal model introduced in Section 5.2 is given in pseudo-code in Algorithm 2. It is defined in two parts, intended to be run in parallel.

The first part is responsible for receiving PDUs. It waits for a PDU to arrive. If this PDU represents a valid transition to the next step of the finite automaton, then this transition is performed. If the next step is an accepting step, the algorithm successfully returns. A critical section is defined using the concept of mutual exclusion, in order to avoid interference between the sending and receiving parts. If the received PDU does not represent a valid transition to the next step, the algorithm aborts.

The second part of the algorithm is responsible for sending PDUs. If valid transitions to the next step of the finite automaton exist having a clock time assigned, the minimum remaining clock time is selected. If no clock time is assigned, a transition is selected at random. After that, the PDU corresponding to the selected transition is sent. The sending part also contains a critical section protected by mutual exclusion. If the current step represents an accepting step, the algorithm returns successfully.

6 Implementation Details

In this section we present two tools and supporting data formats, to enable protocol-based verification within the introduced DCP slave development process. This entire tool chain supports Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), both over Internet Protocol version 4. However, the introduced finite automaton, test procedure and data formats may be used in connection with other DCP-supporting transport protocols in the future as well.

6.1 Workflow

Figure 3 shows a workflow for the DCP test generator and DCP tester tools. The DCP test generator consumes the test scenario template, the test scenario extension, and the slave description for the slave-under-test (1). It produces a test scenario (2), which is handed over to the DCP tester. The DCP tester communicates with the slave-under-test by using a communication system and a chosen transport protocol. It performs the specified test procedure (3). The tester logs all transitions together with timestamps and corresponding PDUs (4). It generates logs that may be used for development or statistical evaluations. The slave-under-test may also generate a report.

6.2 Data Structures

To represent a test procedure an XML schema definition is used. Figure 4 shows a graphical represen-

DOI 10.3384/ecp20174133

input: state: State after which entry the ExtensionSet shall be executed
foreach x *holds* $x \in S \wedge x \notin F \wedge \exists n \in \mathbb{N} : (n, (Receive, NTF\_state\_changed(state)), x) \in \delta$ **do**
  predX $= \{t | t \in \delta \wedge \exists n \in \mathbb{N} : t.from = n \wedge t.to = x \wedge t.\Sigma = (Receive, NTF\_state\_changed(state))\}$;
  sucX $= \{t | t \in \delta \wedge \exists n \in \mathbb{N} : t.from = x \wedge t.to = n \wedge t.\Sigma = (Receive, NTF\_state\_changed(state))\}$;
  entry = MaxStep;
  <execute sub elements>;
  foreach t *in* $predX$ **do**
    t.to = entry;
  end
  foreach t *in* $sucX$ **do**
    t.from = MaxStep - 1;
  end
end

Algorithm 1: Execution of a ExtensionSet

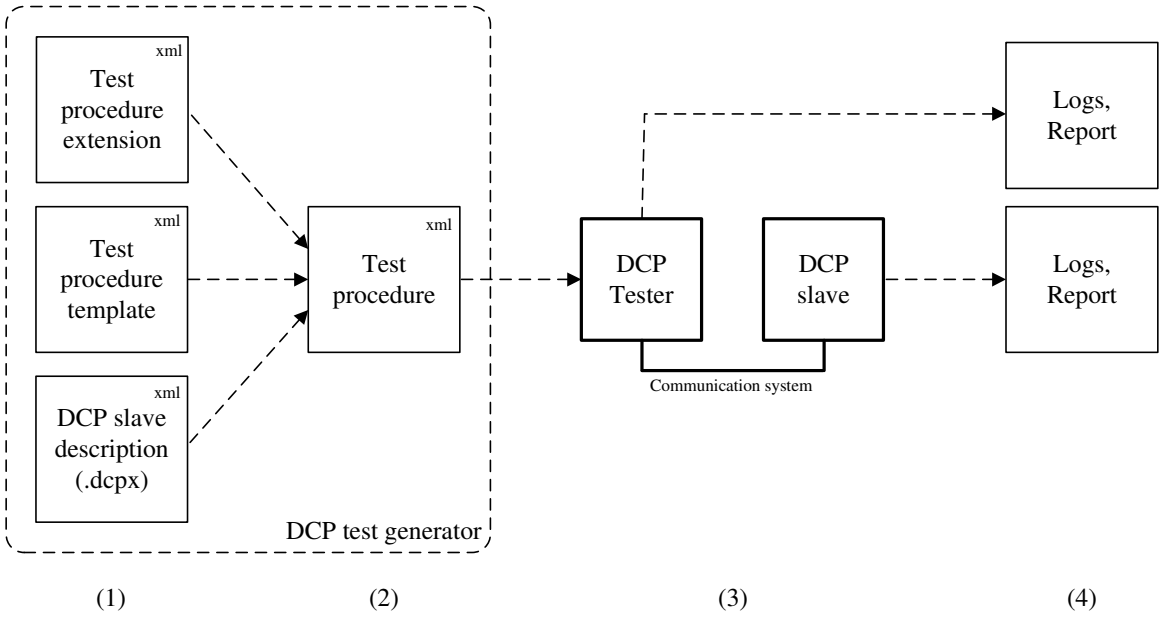

Figure 3. Workflow for DCP test generator and DCP tester tools

tation of this schema definition. It implements the introduced formal model from Section 5.2. A root element `DcpTestProcedure` is defined, which includes the set of sending and receiving `Transitions`. Furthermore, it also contains information for network `Drivers`, i.e., IP and port information for the tester tool.

6.3 Test Generator and Tester as Open-Source Software

The DCP test generator is written in Java. It is implemented as a command-line tool. Its main output is the test procedure. The DCP tester is written in C++. It is implemented as a command-line tool. The test generator[5] and the tester[6] are provided as open-source software, licensed under a BSD

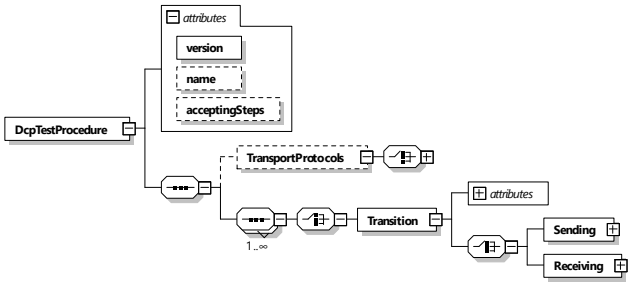

Figure 4. Test procedure data structure as logical view of an XML schema definition.

3-clause license. The DCP specification document is maintained as a Modelica Association[7] Project (MAP). The Modelica Association is a non-profit, non-governmental organization with members from

[5] https://github.com/modelica/DCPTestGenerator
[6] https://github.com/modelica/DCPTester
[7] http://www.modelica.org

```
1  step=0;
2  lastAction=0;
3  lastCheck=now();
4  foreach clocks for every sending transition with defined times stamp in test procedure do
5  |  clock(transition.sending) = transition.sending.numerator / transition.sending.denominator;
6  end
7  do in parallel
8  |  while wait for received PDU AND Receive PDU pdu do
9  |  |  lock();
10 |  |  if ∃ transition ∈ successor(step): transition.receiving = pdu;
11 |  |   then
12 |  |  |  if pdu.log then
13 |  |  |  |  Add PDU to statistics;
14 |  |  |  end
15 |  |  |  step = transition.to;
16 |  |  |  unlock();
17 |  |  |  if isAccepting(step) then
18 |  |  |  |  return 0;
19 |  |  |  end
20 |  |  else
21 |  |  |  unlock();
22 |  |  |  return 1;
23 |  |  end
24 |  end
25 end
26 do in parallel
27 |  while !isAccepting(step) do
28 |  |  lock();
29 |  |  <Update clock time of sending transitions by subtracting time between now and last update>;
30 |  |  transition = null;
31 |  |  if ∃ transition ∈ successor(step): transition.sending is defined AND
   |  |   transition.sending.clockTime is defined;
32 |  |   then
33 |  |  |  transition = <choose transition with min remaining clock time>;
34 |  |  |  if clock(transition.Sending) < 0 then
35 |  |  |  |  clock(transition.Sending) = transition.Sending.numerator /
   |  |  |  |   transition.Sending.denominator;
36 |  |  |  else
37 |  |  |  |  transition = null;
38 |  |  |  end
39 |  |  end
40 |  |  if transition = null AND ∃ transition ∈ successor(step): transition.sending is defined AND
   |  |   transition.sending.clockTime is not defined;
41 |  |   then
42 |  |  |  transition = <choose transition randomly>;
43 |  |  end
44 |  |  if transition != null then
45 |  |  |  <send PDU of transition>;
46 |  |  |  step = transition.to;
47 |  |  end
48 |  |  unlock();
49 |  end
50 |  return 0;
51 end
```

Algorithm 2: DCP tester algorithm

DOI 10.3384/ecp20174133

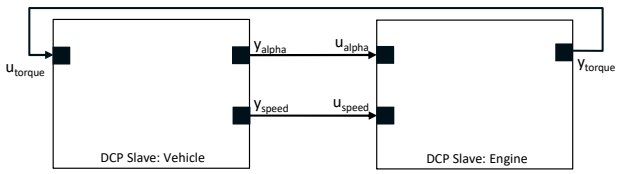

Figure 5. Co-simulation scenario implemented using DCP

Europe, North America, and Asia. Since 1996, its simulation experts have been working to develop the open standard Modelica and the open-source Modelica Standard Library. Today it aims at coordinated standardization, development of software technology, and corresponding methods in the fields of cyber-physical systems and systems engineering. The DCP specification is available as an open-access standard[8] licensed under a Creative Commons BY-SA 4.0 license.

7 Industrial Use Case

7.1 Description

In order to demonstrate the methods and algorithms outlined in this paper, a use case from the automotive domain was used. It consists of a co-simulation scenario including a vehicle simulation and an engine on a testbed. Both were implemented as separate DCP slaves, where the engine model was used as the slave-under-test. This slave has two input variables. The first input represents the accelerator pedal angle measured in degrees, the second represents the shaft speed in revolutions-per-minute. This slave has one output variable representing the shaft's torque in newton metres. It was implemented using standard compliant DCPLib. The intended co-simulation scenario is shown in Figure 5. Our goal is to perform a protocol based test of the DCP slave representing the engine (process level III), before it is functionally tested (process level II) and actually used in context of the shown co-simulation scenario (process level I).

7.2 Results

For this evaluation of protocol-based verification we consider the suggested test procedure template, several different test procedure extensions, and the DCP slave description file of the slave-under-test. First we executed a basic procedure without any extensions at all. The generated procedure passes straight through the different phases of the DCP state machine. Hence, this test procedure can be applied to any DCP slave. Second, the base template was extended to include more protocol tests. The generated procedure tests the slave's capability to deal with valid and invalid values for configuration. Third, a comprehensive extension set including configurations for data exchange was generated. However, all data PDUs for one unique `data_id` are counted only once. This was done to prevent that the same transition in the procedure is verified multiple times. Finally, the third procedure was augmented by including the heartbeat feature (Modelica Association Project DCP, 2019, p.62).

The DCP test generator was used to generate these procedures. The DCP tester was used to execute the test procedures. Table 1 shows the results. All test procedures were successfully evaluated and finalized in an accepting step. The column δ indicates the number of transitions contained in the test procedure, and *LoC* refers to the lines of code in the generated test procedure. Furthermore, the numbers of transitions are divided into numbers of corresponding sent and received PDUs. Based on the execution of the test procedures, we analyzed the numbers of *actually* sent and received PDUs. The last column shows the test procedure execution time, measured on a standard laptop device. All times measured include 4 seconds of simulation time in soft-real-time (SRT) mode. For the communication system, a local UDP socket-based configuration was used.

8 Conclusion

With protocol-based verification we introduce a novel test methodology for distributed co-simulation according to the DCP standard. In the associated DCP slave development process it fits between software- and function-based verification. Continuous protocol-based verification performed on a library designed for standard compliance, e.g., DCPLib, can potentially improve the quality of code over time. Both DCP slave providers and integrators can benefit from protocol-based verification, as they can shift verification activities to earlier phases of system development. The generation of test procedures using steps and transitions is advantageous over linear script based approaches. Extension sets for test procedures provide a scalable way to include new and more specific tests. The DCP test generator and the DCP tester are freely available under open source licenses.

The approach outlined in this paper is not intended as a means for exhaustive testing. A positive test result of protocol-based verification does not confirm that a DCP slave is fault-free and will never violate the specification document. Instead, it provides confirmation that the DCP slave-under-test is able to handle the actually executed sequence of sent and received PDUs. Technology-specific aspects like scheduling, network delay, jitter, etc. may cause non-deterministic behaviour during protocol-based verification.

Some extensions might also depend on the design of the DCP slave-under-test, indicated by e.g., capability flags. So a more modular approach for selection of extension sets could be desirable.

[8] http://www.dcp-standard.org

Test Procedure	δ	LoC	PDUs sent			PDUs received			Time [s]
			Specified	Actual	Ratio	Specified	Actual	Ratio	
Basic (no extensions)	62	325	16	15	0.94	46	37	0.80	10.19
Configuration (no data)	1925	11173	813	673	0.83	1112	695	0.63	10.30
Configuration (including data)	2369	13201	897	722	0.80	1472	747	0.51	12.70
Heartbeat (including data)	9053	49700	2764	742	0.27	6289	773	0.12	14.63

Table 1. Test procedures, their characteristics, and application results.

Acknowledgments

The work reported in this paper was conducted in the course of the ITEA3 Project ACOSAR (N°14004). It was partially funded by the Austrian Competence Centers for Excellent Technologies (COMET) program, the Austrian Research Promotion Agency (FFG), and by the German Federal Ministry of Education and Research (BMBF) under the support code 01lS15033A. The published versions of the DCP test generator and DCP tester tools were contributed by the Simulation & Modeling Group of Leibniz Universität Hannover.

References

Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauss, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neidhold, Dietmar Neumerkel, Hans Olsson, Jörg-Volker Peetz, and Susann Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114, 03 2011. ISBN 978-91-7393-096-3. doi:10.3384/ecp11063105.

Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: Review of methods and relevance for software testing. *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1994*, pages 109–124, 1994. doi:10.1145/186258.187153.

Martin Krammer, Nadja Marko, and Martin Benedikt. Interfacing Real-Time Systems for Advanced Co-Simulation - The ACOSAR Approach. In Catherine Dubois, Francesco Parisi-Presicce, Dimitris Kolovos, and Nicholas Matragkas, editors, *STAF 2016 Doctoral Symposium and Projects Showcase*, pages 32–39, Vienna, Austria, 2016.

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. In *Proceedings of the 50th Computer Simulation Conference*, SummerSim '18, pages 1:1–1:14, San Diego, CA, USA, 2018a. Society for Computer Simulation International. URL `http://dl.acm.org/citation.cfm?id=3275382.3275383`.

Martin Krammer, Nadja Marko, and Martin Benedikt. Requirements engineering for consensus-oriented technical specifications. In *Proceedings - 2018 IEEE 26th International Requirements Engineering Conference, RE 2018*, pages 315–324, Banff, Alberta, Canada, 2018b. ISBN 9781538674185. doi:10.1109/RE.2018.00039.

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, Regensburg, Germany, Feb. 2019. Modelica Association. doi:10.3384/ecp1915787. URL `http://www.ep.liu.se/ecp/article.asp?issue=157%26article=9`.

R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 2002. ISSN 01641212. doi:10.1016/S0164-1212(01)00132-7.

Richard J. Linn. Conformance Evaluation Methodology and Protocol Testing. *IEEE Journal on Selected Areas in Communications*, 7(7):1143–1158, 1989. ISSN 07338716. doi:10.1109/49.44561.

Modelica Association Project DCP. *DCP Specification Document, Version 1.0*. Modelica Association, Linköping, Sweden, 2019. URL `http://www.dcp-standard.org`.

Modelisar Consortium and Modelica Association Project "FMI". Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0.1, 2019.

Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011. ISBN 1118031962.

Steven R. Rakitin. *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, Inc., USA, 2nd edition, 2001. ISBN 1580532969.

Deepinder P. Sidhu and Ting Kau Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989. ISSN 00985589. doi:10.1109/32.16602.