Karsten Albers, INCHRON AG
Benjamin Bolte, itemis AG
Max-Arno Meyer, RWTH Aachen University
Axel Terfloth, itemis AG
Anna Wißdorf, PikeTec

13

Tool Support for Co-Simulation-Based Analysis

The development of collaborative embedded systems (CESs) requires the validation of their runtime behavior during design time. In this context, simulation-based analysis methods play a key role in the development of such systems. Simulations of CESs tend to become complex. One cause is that CESs work in collaborative system groups (CSGs) within a dynamic context, which is why CESs must be simulated as participants of a CSG. Another cause stems from the fact that CES simulations cover various cyber-physical domains. The models incorporated are often managed by different tools that are specialized for specific simulation disciplines and must be jointly executed in a co-simulation. Besides the methodological aspects, the interoperability of models and tools within such a co-simulation is a major challenge. This chapter focusses on the tool integration aspect of enabling co-simulations. It motivates the need for co-simulation for CES development and describes a general tool architecture. The chapter presents the advantages and limitations of adopting existing standards such as FMI and DCP, as well as best practices for integrating simulation tools and models for CESs and CSGs.

13.1 Introduction

Today's heterogeneous engineering tool environments and the rising number of different systems engineering methods lead to the need for tool interoperability. The development of collaborative embedded systems (CESs) adds another factor to the complexity, as the embedded systems involved must be able to work properly in dynamically changing collaborative system groups (CSGs) and within their environment. This leads to more complex development scenarios, as additional methods must be applied to develop these systems and system groups. In addition, more organizations and stakeholders are involved, each potentially using their own modeling methods and supporting tools. In this context, integrating software development tools is a crucial prerequisite for the efficient engineering of collaborative embedded systems. In order to set up an integrated development and modeling approach, two aspects must be covered: first, different methods must be assembled into an integrated methodology; second, interoperability and integration between different tools must be established in order to set up an integrated tool chain. This chapter focuses on the second aspect. While enabling tool interoperability is important for every kind of CES and CSG development method, this chapter focusses especially on enabling tool interoperability for co-simulation-based analysis methods. Enabling interoperability for these kinds of methods is especially challenging, as it requires data integration not only at the level of model artifacts, but also at the level of a joint execution. The focus of this chapter is complementary to Chapter 12, which covers general simulation-based analysis methods.

After categorizing the different kinds of simulation models and motivating the need for co-simulation, we describe a tool architecture that enables co-simulation, together with the relevant standards FMI and DCP. The concepts and approaches discussed are exemplified by the "*Collaborative Adaptive Cruise Control (CACC)*" vehicle platoon use case (see Chapter 1).

13.2 Interaction of Different Simulations

Simulating CESs and CSGs requires the co-simulation of various highly complex models. There are a large number of models that interact

together to provide the functionality of the system under test (SUT) within its context. These include:

- ☐ Environment models (e.g., city with streets and collaborative traffic lights)
- ☐ Behavior models (e.g., CACC, platooning control)
- ☐ Sensor models (e.g., distance sensor)
- ☐ Dynamic models (e.g., vehicle physics)
- ☐ Models for the timing behavior of the execution platforms, the implementation, and the communication
- ☐ CES/CSG interface models
- ☐ Communication models (e.g., wireless car communication)
- ☐ Uncertainty models (e.g., sensor and communication uncertainty)

All these models must be interoperable to enable information exchange and time-synchronized execution. Each simulation tool can execute one or multiple of the models listed.

Let us consider the vehicle platooning use case by way of explanation. A platoon is a collaborative vehicle convoy that uses car-to-car communication based on ad-hoc networks for collaboration. The system that will be used for the platoon control is called Collaborative Adaptive Cruise Control (CACC). This system enables the distance between vehicles to be reduced. The vehicles following the lead vehicle use the data transferred to calculate their relative acceleration.

Platooning use case

The simulation of complete scenarios can be realized by a co-simulation of different tools and model. While these building blocks apply to CESs and CSGs in general, the concrete types of environment and physics models are typically specific to the use case. To evaluate the behavior of the co-simulation participants, it is important that co-simulation results can be reproduced reliably. In general, there is a set of scenarios that are used repeatedly to compare the results of different co-simulations. Since many embedded systems operate in a safety-critical environment, test scenarios might also be prescribed by safety standards. The scenarios and the focus of the analysis will determine which simulated component parts are necessary to meet the test goals. Only some parts of the functional behavior of the target systems need to be included in the specific co-simulation execution and therefore in the underlying models. Other parts can be either substituted (for example, the pre-processing of sensor data) or omitted entirely if they are not needed for the test execution. The selected level of detail for the different model parts will also depend on the test goals. For parts developed by suppliers, the level of detail

Analysis

available for the simulation models can also be limited. More abstract models will increase the test performance and allow test and validation in earlier phases of the development process. On the other hand, the significance and the quality of the test results can be limited for abstract models.

Environment simulation

Fig. 13-1: Co-simulation model platforms and tools

The first building block in [Figure 13-1](#) includes tools capable of simulating the context and environment of the systems under test (SUT), such as the roads on which the vehicles are driving and the interfering traffic. For the example use case, the simulation must cover the individual vehicles of the platoon with their specific movements and distances to each other. It should provide input sensor information from the viewpoint of the systems, such as generated camera images or radar vectors; alternatively, depending on the goal of the simulation and the available or desired degree of detail, the simulation should provide pre-processed data such as distances to objects.

An ad-hoc approach is to use simple step-by-step instructions that give the exact sequence of events in simulation scenarios. However, in a co-simulation with numerous simulation models and simulators interacting with each other, this approach is not very well suited to modeling parallel events that might occur. Another common approach is to use statecharts that are more suited for modeling the interaction and collaboration of the models involved (Section 5.2). Tools that support statecharts for test modeling include the YAKINDU Statechart Tools (YSCT) [Yakindu 2020] and Time Partition Testing (TPT) [PikeTec 2020].

Interactive 3D co-simulations support rapid prototyping scenarios for the development of CESs and CSGs. Therefore, the CESs and CSGs are visualized directly within their environment and interactive changes of system behavior models, as well as environment models, are supported in real time. CARLA [Carla 2020] is a vehicle and

environment simulator tailored for evaluating automated driving functions and therefore especially addresses the vehicle platooning use case. It visualizes the environment, the vehicles and their movements, and the complete traffic scenarios based on Unreal Engine [Unreal 2020]. The view of the environment can be captured by multiple sensors attached to different vehicles. The behavior of vehicles can be influenced and set from other simulation tools. CARLA supports interactive changes to objects of the virtual world in real time to create various scenarios and directly visualize the impact on CESs and CSGs in these varying contexts.

The functional behavior of the CSG can be modeled and simulated with various approaches. MATLAB/Simulink [MathWorks 2020] provides the ability to model and simulate many functions based on sensor, image, or radar data processing. Toolboxes for image processing and autonomous driving functions are available. An exchange with other simulation tools can be provided using co-simulation toolboxes such as the Functional Mock-up Interface (FMI) slave interface. For other scenarios, especially for decision algorithms, modeling the behavior with one or a set of statecharts can be a more suitable approach, and this can be simulated with the YAKINDU Statechart Tools, for example. Another possibility is to include either implemented or generated target code (for example, in C++) in the co-simulation.

Functional simulation

Special simulation tools, such as chronSIM [INCHRON 2020a], augment the co-simulation by incorporating the timing behavior of the software, the execution platforms, the scheduling effects, and the communication between and within systems. In particular, timing effects and delays of the complete event chain, from the sensors, through the processing, to the actuators are derived (see [Figure 13-2](#)). The timing simulation replicates the timing of CESs implementing the CSG. Based on the models of the systems and the software, the simulator calculates resulting delays, end-to-end delays for the data processing, as well as potential data losses and more.

Timing simulation

provided by Expleo, xMOD provided by FEV, and TPT provided by PikeTec. Custom implementations can also add co-simulation platform features to tools like CARLA.

13.3 General Tool Architecture

This section provides details of a proper tool architecture, co-simulation standards, and their application to the CES and CSG simulation.

The need for a time-synchronized execution naturally leads to a master-slave architecture. This architecture defines two roles for tools participating in the co-simulation: the *co-simulation master* and the *co-simulation slave*. The *master* manages a set of *slaves*, coordinates interaction between them, handles time synchronization, and makes co-simulation results accessible for subsequent analysis steps. The *co-simulation slave* provides a simulation API (application programming interface), which is used by the co-simulation master to proceed with the simulation, together with a description of the slave's functional interface, such as signals that are consumed or produced by other slaves.

*Master-slave
architecture*

This description of slave interfaces forms the basis for the configuration of a *co-simulation*. Engineers have to specify the mapping between the interfaces of the slaves involved to realize the intended data flow between models. In addition, the simulation duration and time step for updating the simulation models are defined according to the requirements of the co-simulation participants involved. This is necessary to achieve the required accuracy during the simulation.

*Co-simulation
configuration*

This configuration can be defined using a *co-simulation configuration service*, which can be either a tool with a UI or an automated service that applies a transformation from an existing system or CSG model that contains the required information. In any case, it is the co-simulation master's obligation to process this configuration correctly.

As the co-simulation of many co-simulation components can be resource-intensive and time-consuming, especially when using real-time 3D rendering, parallelization of the co-simulation components over multiple processing units is beneficial to provide more computing resources and to decouple components such as the co-simulation master, modeling tools, and the visualization. This does not

*Distributed co-
simulation*

mean that the tools must be used in a distributed computing context; they can also run on one computer without any distribution.

Fig. 13-3: Co-simulation tool architecture

Tool interoperability

Figure 13-3 shows the exemplary architecture of an interactive co-simulation. To enable a tool platform to support co-simulation, various tools and models must be made interoperable. We identified two standards as particularly relevant in the context of developing CESs and CSGs. The first is *Functional Mock-up Interface (FMI)* and the second is the *Distributed Co-Simulation Protocol (DCP)*. These standards can be applied by tool developers as a basis for setting up co-simulation features or in combination with existing proprietary solutions to extend tool interoperability. Both standards comply with the main architectural principles and will be introduced in subsequent sections.

13.4 Implementing Interoperability for Co-Simulation

FMI standard

The FMI for co-simulation standard [Modelica Association 2019a] addresses the integration of heterogeneous simulation models and tools that match the existing constraints for the development of CESs and CSGs. It defines the required technical master-slave interface for a master-slave architecture.

Each model is provided by a co-simulation slave called a Functional Mock-up Unit (FMU). An FMU is a zip file containing at least one executable binary library, along with an XML file that includes the interface definition for the slaves. Libraries for multiple platforms can be included to support portability. The FMI co-simulation master dynamically loads and executes the binary libraries of all slaves. The master-slave interface itself is defined as an API in the C programming

language, with an underlying state machine that defines the order of interface calls.

The FMU interface concept is based on a data-flow paradigm. A model defines a set of input and output variables with simple data types, such as real, integer, and string. The FMI master proceeds with the simulation step by step. In each step, all FMUs are provided with the current input values and are then executed. Finally, the output variables are propagated to the input values for the next execution cycle. Thus, data exchange occurs only between successive execution steps.

FMI brings with it some constraints that may be relevant when considering the FMI standard compared to proprietary approaches. *Structured data types* do not have a direct counterpart in FMI but must be substituted by simple variables, which flattens the hierarchical data structure. *Events* can only be mapped to changes of input or output values, such as rising and falling edges, which requires the application of conventions between all co-simulation participants. The same is true for synchronous *operation* calls, which would require a complex protocol consisting of call and return events. Finally, *behavioral types* supporting dynamic reconfiguration in CSGs cannot be mapped in a meaningful way.

FMI limitations

The co-simulation master controls the simulation progress and is thus responsible for the time synchronization between all co-simulation components involved. In FMI, each FMU implements the *fmi2DoStep* function which gets the current simulation time point and the duration of the next time step as parameters. The FMI master decides on the step size, which can be of fixed or variable length. The slaves proceed with the simulation for the requested step size. Slaves can use a virtual clock to provide faster than real-time executions, which is relevant for long-running simulations or repeated test scenarios.

Time synchronization

Co-simulation participants, such as visualization and physics engines, might provide their own timing behavior that must be synchronized. The first possibility is to use an external co-simulation master to set the timing. Therefore, slaves (e.g., CARLA) must provide a time synchronization interface. Second, if only one additional tool with its own timing is used, this can be extended by a custom implementation of a co-simulation master.

To cope with the standard, for each participating simulation model, a *co-simulation slave generator* transforms the model into a standalone executable co-simulation slave. The code generation typically involves the generation of a code layer that implements the

Co-simulation slave generator

required functionality for data exchange and time synchronization according to the simulation API realization used (e.g., FMU Interface) so that co-simulation-slaves act as a black-box component for the co-simulation master. Code generation can be used to support different software and embedded hardware platforms to reduce manual implementation efforts and to improve quality. Figure 13-4 shows how the co-simulation slave generation concept is applied for creating FMUs.

Fig. 13-4: Building FMUs using C-code generation

The generator derives the meta-information description from the model's interface definition to describe input and output variables. In addition, the code for the model execution library is generated, which consists of code to access or implement the executable model and an adapter for the FMU simulation API. If a modeling tool already provides a code generation or interpreter for its models, then it is good practice to reuse these and just add a generator for the required adapter code. Finally, the model description and executable library are bundled as an FMU zip file.

13.5 Distributed Co-Simulation

Distributing models in a co-simulation across multiple platforms is another key interest for realizing complex simulations for CESs and CSGs. A communication infrastructure for connecting the different co-simulation participants is required. While distribution is a proposed FMI use case, the realization of distribution and the communication layer are left untouched by the standard.

*Co-simulation tool
wrapper*

If an external tool is required for simulating a slave model, a direct execution within the *FMI master* is not possible as it assumes the complete execution of the co-simulation within a single multi-threaded process. A *co-simulation tool wrapper* is a specific *co-simulation slave* that, instead of executing the model itself, delegates all execution requests to the external simulation tool. This requires a communication layer that must handle data exchange, time synchronization, and invocation of simulation. Additionally, the co-

simulation slave must adapt its simulation API to the communication layer and handle data binding. The standard does not prescribe the concrete communication protocol and therefore any existing protocol can be reused.

To overcome the need for a tool-specific wrapper, the Distributed Co-Simulation Protocol (DCP) was developed as an open, accompanying standard to FMI [Krammer et al. 2018]. It standardizes the distribution of models on different software and hardware platforms, which is particularly important for handling models bound to specific execution environments and to increase simulation performance. DCP focusses particularly on the following aspects. First, DCP is specifically designed for the integration of real-time and non-real-time systems simultaneously. Therefore, it is possible to perform co-simulations that combine hardware setups with digital models. Second, DCP can be combined with other standards, such as FMI and proprietary solutions. Third, DCP supports a wide range of communication protocols (UDP, TCP, CAN, USB, Bluetooth 3) to ensure interoperability on the application layer regardless of the communication medium [Modelica Association 2019b].

DCP also applies a master-slave architecture. As in FMI, DCP requires XML-based configuration data and defines a state machine for the execution and communication life cycles of slaves. The communication to the master and other slaves is handled via protocol data units (PDU) defined by DCP. Thus, the specification provides a precise basis and guidance for the implementation of DCP on the master and slave side by tool developers.

The option to flexibly distribute master and slave to different hardware and operating systems can be used effectively in CES and CSG co-simulations to increase overall simulation performance and to integrate interactive simulations or concrete hardware. In particular, DCP enables synchronization in either real time or non-real time with other non-DCP co-simulation slaves, such as plain FMUs or a co-simulation tool wrapper using proprietary protocols.

The initial implementation effort for DCP is definitely higher compared to reusing an existing proprietary protocol. However, once implemented, masters and slaves can interoperate natively with other platforms that support the standard. With regard to modeling concepts, DCP has the same limitations as FMI.

Distributed Co-Simulation Protocol - DCP

DCP limitations

13.6 Analysis of Simulation Results

During and after the execution of the simulations, a co-simulation analysis service is necessary to evaluate and extract the simulation results for conclusions and follow-up decisions.

For certain scenarios, the success can be determined by checking whether the co-simulation participants reach/avoid a failure state or meet predefined goals. The different simulators and tools can track such conditions directly during the simulation execution.

For evaluations that cannot be executed directly in the co-simulation platform, a useful approach is to record the execution and system states in one or a set of trace files during the simulation. Based on the information thus gathered, the fulfillment of requirements can be checked and statistical information on the behavior can be derived.

Analysis and reporting tools read the machine-readable simulation traces for further processing. The use of open formats can help with processing of simulation traces from a larger set of tools in a co-simulation. The information available from the different simulations will be quite different, which will affect the required trace formats. For example, state transitions for the state machines can be recorded in a behavior model, allowing the engineers, for instance, to validate models on an even deeper layer and enable gray and white-box verification. For the timing behavior, execution states, events, and data processing chains should be recorded. A synchronization of the trace files from the different simulation tools is necessary for the evaluation of cross-over aspects. Time stamps from the common time base or shared frequent events can make synchronization easy.

Another advantage of traces is that they make it easier to determine the reasons for an observed behavior, such as a failed requirement. Critical situations can be visualized and explored by tools like *chronVIEW* [INCHRON 2020b].

13.7 Conclusion

In this chapter, we addressed the task of enabling tool interoperability for co-simulation-based analysis methods for CESs and CSGs. A particularly challenging aspect for enabling tool support for co-simulations is that the tool integration must facilitate a joint execution of model artifacts that are integrated at a data level.

A distributed master-slave architecture with well-defined interfaces is the basis for orchestrating and coordinating heterogeneous models and tools into a co-simulation. The FMI and

DCP standards support this architecture. FMI-compliant models can be reused and executed on different co-simulation platforms, which may serve a specific purpose. DCP-enabled platforms and tools can easily be connected. Both standards can be combined with existing proprietary solutions, enabling reuse of simulation tools, platforms, and communication infrastructure.

The data-flow-oriented approach of FMI and DCP has limitations with regard to applicable modeling concepts and this constrains the applicability for co-simulation scenarios that require dynamic reconfiguration of CSGs. Here, proprietary approaches may be a better fit. The standards also do not define a model for connecting slaves. This is the responsibility of the concrete master implementations. Thus, CSG models that describe such model relationships must be mapped specifically for each master implementation and are not easy to reuse.

The distributed setup enables integration of heterogeneous co-simulation tools, which may even support interactive changes to the models during runtime. As a result, the development process can potentially be improved in certain ways. First, an explorative development of models without time-consuming code generation steps is provided. Second, many functional components from various vendors can be combined for rapid prototyping and early testing scenarios. Third, the visualization of test scenarios has potential to improve the communication with the stakeholders involved across various organizations.

Co-simulation improves verification and validation of CESs and CSGs. Trace information from all co-simulation participants enables required analysis methods and tools to enhance verification and allow a statistically rich evaluation. Simulation tools contribute environment, function, timing, uncertainty, and physical models in a scope and a level of detail that is appropriate for different scenarios to the co-simulation. The resulting co-simulation thus better reflects real-world scenarios, which improves the generalizability of the validation and verification results.

13.8 Literature

[Carla 2020] CARLA – Open Source Simulator for Autonomous Driving Research: <https://carla.org>, accessed on 05/08/2020.

[Expleo 2020] MESSINA – Test Automation and Virtual Validation for Embedded Systems: <https://www.expleo-germany.com/en/produkte/messina/>, accessed on 05/08/2020.

- [INCHRON 2020a] chronSIM – Model-Based Simulation of Embedded Real-Time Systems: <https://www.inchron.com/tool-suite/chronsim>, accessed on 05/08/2020.
- [INCHRON 2020b] chronVIEW <https://www.inchron.com/tool-suite/chronview/>, accessed on 05/08/2020.
- [Krammer et al. 2018] M. Krammer, M. Benedikt, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner, M. Damm-Norwig, V. Schreiber, N. Nagarajan, I. Corral, T. Sparber, S. Klein, J. Andert: The Distributed Co-Simulation Protocol for the Integration of Real-Time Systems and Simulation Environments. In: Proceedings of the 50th Computer Simulation Conference, 2018.
- [MathWorks 2020] Simulink – Simulation and Model-Based Design <https://www.mathworks.com/products/simulink.html>, accessed on 05/08/2020.
- [Modelica Association 2019a] Modelica Association Standard: FMI – Functional Mock-Up Interface Specification Document 2.0.1, 2019.
- [Modelica Association 2019b] Modelica Association Standard: Distributed Co-Simulation Protocol (DCP). Specification Document 1.0.0, 2019.
- [PikeTec 2020] TPT: Control Testing Made Easy <https://piketec.com/tpt/>, accessed on 05/08/2020.
- [PikeTec 2020b] Automatic Test Case Generation in TPT <https://piketec.com/tpt/testcase-generation/>, accessed on 05/08/2020.
- [Unreal 2020] UNREAL ENGINE <https://www.unrealengine.com>, accessed on 05/08/2020.
- [Yakindu 2020] YAKINDU Statechart Tools <https://www.itemis.com/en/yakindu/state-machine/>, accessed on 05/08/2020.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.