2017 43rd Euromicro Conference on Software Engineering and Advanced Applications

Rapid Construction of Co-simulations of Cyber-Physical Systems in HLA using a DSL

Thomas Nägele
Radboud University
Nijmegen
The Netherlands
t.nagele@cs.ru.nl

Jozef Hooman
Radboud University & TNO-ESI
Nijmegen & Eindhoven
The Netherlands
hooman@cs.ru.nl

***Abstract*—The development of cyber-physical systems (CPSs) is a multi-disciplinary process. A model-based approach during the design of a system is important for making design decisions during the exploration of alternatives. However, all disciplines use different modelling tools and techniques, which makes the integration of these models difficult and time-consuming. The use of the High Level Architecture (HLA) simplifies this problem, but still requires quite an effort to implement. Our work focuses on minimising the effort required to construct co-simulations. We have created a Domain Specific Language (DSL) to define a system design consisting of different types of models. We demonstrate how this DSL can be used to experiment with alternative designs of the system quickly. The DSL allows us to build virtual prototypes of CPSs without the large overhead of constructing the co-simulation.**

***Index Terms*—Domain Specific Language, Cyber-physical systems, Co-simulation, HLA, FMI**

I. Introduction

The development of cyber-physical systems is a complex process in which multiple disciplines collaborate. Disciplines follow a model-based development approach for the design of specific subsystems. However, most disciplines use their own modelling language and tool. These different modelling techniques make it rather difficult to connect these models into one co-simulation that can be used to simulate the entire system. To overcome this problem, several co-simulation frameworks have been developed, including the High-Level Architecture (HLA) [1]. HLA is a general purpose architecture standard for co-simulation of models from different tools. It can also be used for distributed co-simulation.

However, constructing a co-simulation for a system in HLA is rather complex. All simulation models require a wrapper to be properly embedded in the co-simulation. Additionally, the HLA framework requires a Federation Object Model (FOM), which is an XML document describing all objects, interactions and attributes in the co-simulation execution. Maintaining the FOM is rather time-consuming if the system is changed frequently, for instance during design space exploration. Consequently, co-simulation is often not used on a regular basis in industry. The effort required for the setup and maintenance of the co-simulation becomes a blocking factor. Such a situation should be avoided by minimising this effort.

We aim for a method that enables the rapid construction of an HLA-based co-simulation. In particular, we intend to incorporate models of software in the co-simulation. The method should be capable of quickly connecting multiple models created in different tools while requiring only little effort from the system designers. Changes in the system architecture, addition of loggers or the construction of multiple different configurations of one system should be implemented with little effort. This enables fast design space exploration using virtual prototypes of the system. At all times, it is important that the time footprint for the system designers is as small as possible.

A. Approach

In [7], we suggested a model-based method to create an HLA-based co-simulation of different types of models. Our goal was to include separate models of the software in the co-simulation to stimulate concurrent development of both software and hardware. This method uses HLA to co-simulate continuous-time models with discrete-time models of software. However, the connection of these types of models still requires significant manual labour. To speed up the implementation for a couple of simulator types, we improved the co-simulation construction.

First, we developed connection libraries for these simulators. These libraries offer methods to control attribute values and time in the simulators and are focused on reusability. A description of the used techniques can be found in Section II.

Second, we use a Domain Specific Language (DSL) for the configuration of the co-simulation. The DSL is used to specify the objects, attributes, interactions and connections between the objects in the system. From this specification, code is generated for the co-simulation. Some default configuration parameters may be set in the DSL to enable the generation of a pre-configured co-simulation implementation. The system specification in our DSL together with the models of the subsystems is sufficient to start the co-simulation of the system. The DSL we developed is discussed in Section III.

B. Related work

Several works on enabling rapid co-simulation development have been published. Neema et. al. [5][8] have already demonstrated an approach to integrate multiple FMUs [3] into one HLA co-simulation in which FMU containers are

978-1-5386-2141-7/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAA.2017.29

automatically wrapped as federates. In their C2WT[1] project, they also connect different types of models together by means of an HLA runtime. This project, however, concentrates on the co-simulation of tools that are being used for their particular wind tunnel application. Defining a co-simulation also requires quite some effort, because a number of meta-models should be specified for this. Since at least one of our modelling tools for software (POOSL) does not fit in this approach, the C2WT framework is less suitable for us.

Another tool that is capable of creating model-based co-simulations of systems is SimGE[2] [11]. SimGE can be used to design a co-simulation federation and is capable of generating federate code. The generated code, however, is limited to skeleton code and therefore SimGE can be very useful when building a full-size HLA federation implementation, but still requires some manual implementation steps.

The INTO-CPS project [6] also aims to improve the multi-disciplinary development of CPSs. They have created an integrated tool chain that supports co-simulation of models, including HiL and SiL simulations. Their approach covers the entire development process, starting with the requirements and ending with actual software and hardware. INTO-CPS uses the FMI standard to support heterogeneous co-simulation of different models and provides a Co-simulation Orchestration Engine (COE) for managing the co-simulation. A consequence of using this integrated tool chain for the development of CPSs is that it makes the development process heavily dependent on the tool chain. We intend to use existing standards to increase the flexibility of this process.

Ptolemy II [9] is another framework that can be used for heterogeneous co-simulations of models. The framework, however, does not rely on existing standards, which also affects the flexibility of the project.

II. Co-simulation tooling

We use a number of techniques and standards for the construction of the co-simulation of a system. This section outlines the most important ones.

A. HLA

The High-Level Architecture (HLA) [1] is an architecture specification for distributed co-simulation. It consists of one Run-Time Infrastructure (RTI) that manages all connected simulations. These simulations together with the RTI are called a federation. Each federation consists of a set of simulation entities, which are called federates in HLA. The HLA standard consists of a set of methods that all simulation entities should implement, an object model template and a set of rules to ensure compliance with the interface. The interface specification covers, amongst others, data distribution management and time management.

Data distribution in HLA is supported in a publish-subscribe structure. A federate may specify its published attributes to the RTI and subscribe to a number of attributes from the other federates. Updates of attributes are pushed to the federates that have subscribed to this attribute by the RTI. Interactions in HLA work in a similar way, but can be considered as labeled broadcasted messages containing zero or more attributes. Each federate that is subscribed to a broadcasted interaction will receive the interaction.

In HLA, each federate follows its own timing behaviour. A federate may be time regulating, time constrained, both or neither one of these. A time regulating federate is capable of sending time stamped messages, while a time constrained federate can receive them. Each federate requests a time advance of a given step size to the RTI and then waits for a time advance grant. If a federate is time regulating, the federate also has a lookahead value. The lookahead value represents a time period during which the federate shall not send any updates. Updates from the federate will at least have a timestamp of the current time plus the lookahead value of the federate. The RTI grants time advance requests to federates that have received all possible updates from other federates. Note that there is no global notion of simulation time in the federation, as each of the federates has its own.

Both commercial and open source implementations of the HLA standard are available. These implementations usually offer an RTI together with a set of interfaces in Java, C or C++. Some modelling tools, such as the MATLAB HLA Toolbox[3], offer integrated HLA support. We used the PoRTIco[4] HLA implementation in our previous experiments [7], but switched to OpenRTI[5] as PoRTIco did not properly implement the *next-MessageRequest* method and was not very actively maintained.

OpenRTI is an open source HLA implementation bundled with an RTI. It offers language bindings for both C++ and Python to build federates. OpenRTI supports all features that we need and is maintained on a regular basis.

B. POOSL

To create models of control software, we use the Parallel Object-Oriented Specification Language (POOSL) [10]. POOSL is a formal language which allows modelling of complex software architectures, including timing aspects. The POOSL tooling[6] allows simulation and debugging of models. The POOSL IDE is offered as a plug-in for Eclipse[7]. The models created in POOSL can be simulated in the Rotalumis simulator[8]. Since we already have large POOSL models of industrial software architectures, we are interested to simulate these models together with models of hardware components. POOSL is also relatively easy to learn.

To be able to connect a POOSL model simulation to the HLA co-simulation, we developed a C++ library that can be connected to the debugging socket of Rotalumis.

[1] https://wiki.isis.vanderbilt.edu/OpenC2WT/
[2] https://sites.google.com/site/okantopcu/simge
[3] http://www.forwardsim.com/
[4] http://www.porticoproject.org/
[5] https://sourceforge.net/p/openrti/wiki/
[6] http://poosl.esi.nl/
[7] https://eclipse.org/
[8] http://www.es.ele.tue.nl/poosl/Tools/rotalumis/

The library implements a simulator wrapper that supports time control and attribute management. This simulator can be connected to the RTI by wrapping it in a federate implementation according to the HLA standard. Figure 1 depicts how the Rotalumis simulator is connected to the RTI through the library.

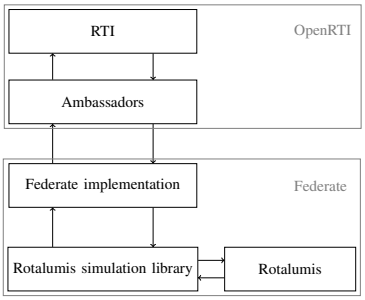

Figure 1. Connection of a single Rotalumis simulation to the RTI.

C. FMI

The Functional Mock-up Interface (FMI) [3] is a standard for model exchange and co-simulation. The FMI standard specifies interfaces to support the exchange of models and to enable co-simulation of dynamic models as so-called Functional Mock-up Units (FMUs). FMI does not specify how FMUs can be combined into a coherent distributed simulation environment. To achieve this, we use HLA.

The FMI standard is widely supported by modelling tools, which allows us to support the connection of models from many different tools to HLA by using a single library. The library setup is very similar to the library we created for connecting Rotalumis to HLA as displayed in Figure 1.

D. Logging

Loggers are often used to save the state of a system while running. To provide similar functionality for our co-simulation, a logger library was developed. This library can join an OpenRTI execution as a federate and subscribes to all attributes from other federates that it records. The logger stores these attributes and requires an end time that specifies when the logger should stop storing these values. When the end time is reached, the logger will save all values as a table in a CSV (Comma-Separared Values)[9] file.

III. Co-simulation definition using a DSL

This section describes the development of a method to quickly construct a co-simulation of different models using a DSL. The system designer can use the DSL to decompose a system into subsystems, each with its own model. Each subsystem is described as an object having a number of attributes. Also, the simulator type for each of the models and their connections with other models can be specified in the DSL. This system specification together with each of the simulation models is enough to construct a fully working co-simulation in OpenRTI. Section III-A describes a sample system, after which Section III-B provides some details on the definition and implementation of the DSL. Section III-C describes how we can define the sample system in our DSL and Section III-D discusses some results and experiments.

A. Example: RoomThermostat

To illustrate our approach to construct a co-simulation, we will discuss a thermostat system system in a house. The house consists of a number of rooms and a thermostat that controls the heaters in the rooms. The co-simulation consists of three different models of rooms and one model of a thermostat.

The three models of the rooms are continuous-time models created in 20-sim[10]. Each of these models has one input and one output. The 'heaterState' is a boolean input attribute that represents the state of the heater in the room. The output attribute of the model is the current temperature.

The model of the thermostat is a discrete-time model created in POOSL that implements a simple switching algorithm for determining the heater state. The thermostat checks its input attribute 'temperature' once every 30 seconds and sets the output attribute 'heaterState'. The heater state is turned on or off (`true` or `false`) when the temperature drops below or rises above the target temperature with a threshold of 1.5%. The predefined target temperature is $20°C$.

We shall refer to the co-simulation of all these models as RoomThermostat. Figure 2 shows the connection layout of the different models in the simulation. Note that the values of the 'temperature' attribute of the federates Bedroom and Hall are not used in the co-simulation.

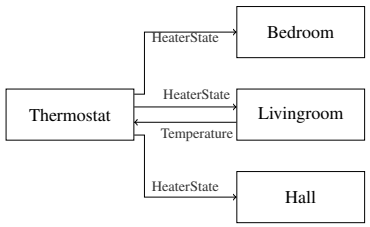

Figure 2. Connection layout for the RoomThermostat co-simulation.

B. The DSL

To decrease the implementation effort required to construct a co-simulation, we developed a Domain Specific Language (DSL) to easily specify a co-simulation and to automatically generate co-simulation code for it. The DSL framework consists of two components: the grammar and the code generation. Our DSL grammar is defined in Xtext[11] [4] using the Eclipse framework. Xtext allows us to define a DSL and generate a full infrastructure for it, including IDE support in Eclipse.

An instance of our DSL starts with the definition of the environment to generate for, such as the type of RTI to use and the locations of external libraries. Then, a number of federate classes may be defined, each of them having a number of attributes. Optionally, federate classes may have predefined default parameters, such as the location of the model and the step size to use during the execution. Finally, a co-simulation definition is provided, consisting of a number of federate instances and a set of connections. The federate instances are

[9] `https://tools.ietf.org/html/rfc4180`

[10] `http://www.20sim.com/`

[11] `http://www.eclipse.org/Xtext/`

instances of the federate classes as defined before. Connections are connections between attributes of federate instances.

The code generation is implemented using Xtend[12], which is a dialect of Java. Xtext and Xtend provide powerful tools for defining and implementing a custom DSL, as described in [2]. From a system definition in the DSL, we generate a CMake[13] project for co-simulation of the system. Currently, only code generation for OpenRTI is supported.

Our code generation generates a set of source files in an organised directory structure. The generated code consists of two parts. The first part is the actual federate, simulation and execution code for HLA together with the FOM and CMake file that ties the project together. The generated code for the first part consists of the following files.

- A FOM XML containing the HLA federation description.
- A CMake file to specify the project for the CMake tool.
- A C++ simulation wrapper for each federate class.
- For each federate instance the following files:
  - A C++ federate implementation
  - A C++ file containing the federate's main method

These sources include code for the detection of other federates that may join the federation. Methods to publish and subscribe to attributes of other federates are generated according to the connection specification in the federation definition in the DSL. Also, attribute connections are realised in the generated code by handling attribute updates from subscribed attributes. For the RoomThermostat, the source DSL containing 86 lines of code expanded to 898 lines of C++ code for the federate classes, instances and simulators and an XML FOM of 207 lines.

The second part consists of additional scripts that are being generated. These scripts provide an easier method to build and execute the co-simulation, but also to override some of the default parameters of the co-simulation, such as the model that a federate should run, step size or lookahead that may have been specified in the DSL. The scripts increase the flexibility of the co-simulation configuration and allow batch execution of the co-simulation through another script. Such a batch script would support the implementation of automated requirement verification based on the system definition and its models and can also be useful during design space exploration.

C. DSL instances

This section demonstrates how the RoomThermostat co-simulation can be constructed in our DSL. Listings 1, 2 and 3 together form a large part of the definition of the co-simulation.

Listing 1 displays the specification of the HLA environment we want to build the project for. The HLA version is set to OpenRTI and the class path is used to specify the directory name where the sources should be generated and directories to find the library and header files of our simulation libraries.

Listing 2 displays a partial domain model definition of our RoomThermostat. Here, we specify the Room federate class

[12]http://www.eclipse.org/xtend/
[13]https://cmake.org/

```
1 HlaEnvironment {
2  HlaVersion OpenRTI
3  ClassPath {
4   src "src"
5   lib "/opt/ORTI-lib/lib"
6   inc "/opt/ORTI-lib/include"
7  }
8 }
```

Listing 1. HlaEnvironment definition for our RoomThermostat.

that may join our co-simulation. The federate class must have specified a TimePolicy, which can be Regulated, Constrained, RegulatedAndConstrained or None, corresponding to the time policies as defined in HLA. Lines 6 to 11 specify the attributes for the Room federate. Each attribute requires a data type. A federate class must also have set a simulator type, which is the simulator that runs the models. The type may also be a CSV logger as described in Section II-D.

The default step setting on line 14 specifies whether the federate should send a *nextMessageRequest* or a *timeAdvanceRequest* to the RTI to advance in time. Additional default parameters can be set, as shown on line 15. Without any further specification, each federate assumes its model to have a name equal to the name of the federate instance. For example, a federate instance named 'Toilet' with the simulator type FMU or POOSL will simulate the model 'Toilet.fmu' or 'Toilet.poosl' respectively by default. This default setting can be overridden by specifying a default model.

```
1  DomainModel {
2   FederateClasses {
3    FederateClass Room {
4     TimePolicy RegulatedAndConstrained
5     Attributes {
6      Attribute temperature {
7       DataType Real
8      },
9      Attribute heaterState {
10      DataType Boolean
11     }
12    }
13    SimulatorType FMU
14    DefaultStep Message
15    DefaultStepSize 30
16   }
17  }
18 }
```

Listing 2. Domain model for our RoomThermostat.

Listing 3 displays how the RoomThermostat co-simulation can be specified in our DSL. The federation consists of a number of instances of the federate classes defined in the domain model. For our RoomThermostat, we define the three rooms – Livingroom, Hall and Bedroom – which are all instances of the federate class 'Room' (not shown here), and a thermostat called LivingroomThermostat, which is an instance of the federate class 'Thermostat'. The attributes of these federate instances can then be connected to each other to share their attributes. Attribute sharing is defined in our DSL by connecting these attributes using the notation `instance1.attribute1 <- instance2.attribute2`, which means that each update of 'attribute2' from 'instance2' will be stored to 'attribute1'

of ‘instance1’. In our example, we connect the heater state as published by the LivingroomThermostat to all models of the rooms and only use the temperature in the Livingroom as input.

```
Confederation House {
 Instances {
  Instance Livingroom as Room,
  Instance Hall as Room,
  Instance Bedroom as Room,
  Instance LivingroomThermostat as Thermostat
 }
 Connections {
  Connection { Livingroom.heaterState <-
       LivingroomThermostat.heaterState },
  Connection { Hall.heaterState <-
       LivingroomThermostat.heaterState },
  Connection { Bedroom.heaterState <-
       LivingroomThermostat.heaterState },
  Connection { LivingroomThermostat.temperature
       <- Livingroom.temperature }
 }
}
```

Listing 3. Confederation definition for our RoomThermostat.

This approach allows easy changes of the number and type of federates co-simulated in the HLA federation. It also provides a method to change the federate connections.

D. Results

The power of our DSL lies in the possibility to make quick changes to test different co-simulation configurations. To illustrate this, we add a new instance of a thermostat. This instance uses the existing model of the thermostat and will be connected to the Bedroom federate. To achieve this, one new instance and one new connection were added and we modified one old connection. Apart from the time to open the project, this change took less than a minute to complete and enabled us to run a new co-simulation without having to implement any new functions. To gain insight in the temperatures of the rooms, we added a logger to the co-simulation. Figure 3 displays the results of both the original configuration and the extended configuration. The charts ‘Livingroom’, ‘Hall’ and ‘Bedroom’ show the temperature in the rooms with the original configuration, while the chart ‘Bedroom extended’ displays the extended configuration. For the latter configuration, the other charts remain untouched. The results are as expected, as the temperature in the Bedroom now fluctuates around 20°C.

IV. Conclusion

We have presented a method to quickly construct a co-simulation of cyber-physical systems using a DSL. The method integrates nicely in the development process, as it only requires one system definition to connect existing models. In Section III-D, we demonstrated how our DSL can be used to experiment with the design of a system. The method does not require any change in the modelling tools that are being used and can therefore be very useful during the design space exploration for a system.

In future work, we intend to explore the possibilities to use our DSL for the verification of system design. For this, automated unit tests for a model-based system design approach

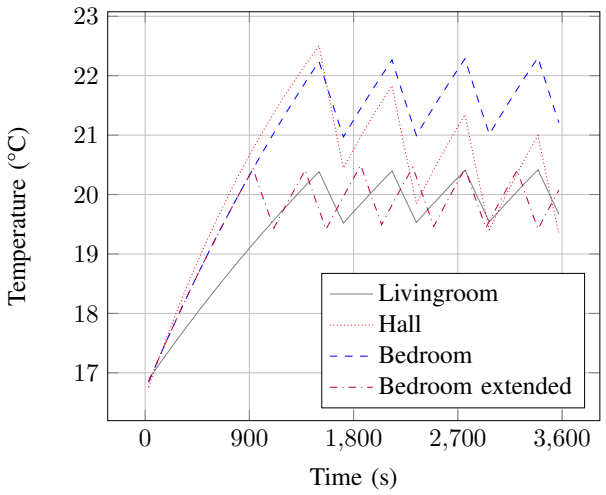

Figure 3. Room temperatures during RoomThermostat co-simulation.

could be very useful. Another DSL could be used to define a set of requirements the system should meet, after which a nightly simulation run can be started that checks whether the system complies to the requirements.

We will also use our method for design space exploration in an industrial context, where case studies will be used to improve our model-based system design approach.

Acknowledgements

We would like to thank Tim Broenink from the University of Twente for creating the models of the rooms.

References

[1] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010*, pages 1–38, Aug 2010.

[2] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[3] T. Blochwitz, M. Otter, et al. The functional mockup interface for tool independent exchange of simulation models. In *8th Modelica Conference*, pages 105–114, 2011.

[4] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[5] G. Hemingway, H. Neema, et al. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, 88(2):217–232, 2012.

[6] P. G. Larsen, J. Fitzgerald, et al. Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6, April 2016.

[7] T. Nägele and J. Hooman. Co-simulation of cyber-physical systems using HLA. In *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual*, pages 1–6. IEEE, 2017.

[8] H. Neema, J. Gohl, et al. Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In *Proceedings of the 10th International Modelica Conference*, number 96, pages 235–245. Linkping University Electronic Press, 2014.

[9] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[10] B. D. Theelen, O. Florescu, et al. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *5th Conference on Formal Methods and Models for Codesign*, MEMOCODE ’07, pages 139–148. IEEE Computer Society, 2007.

[11] O. Topçu, L. Yilmaz, H. Oğuztüzün, and U. Durak. *Distributed Simulation – A Model Driven Engineering Approach*. Springer, 2016.