INtegrated TOol chain for model-based design of CPSs

INTO-CPS Maestro Documentation

Version: 0.01

Date:

The INTO-CPS Association

http://into-cps.org

Contributors:

Casper Thule, Aarhus University, Centre for Digital Twins
Kenneth Lausdahl, Mjølner Informatics A/S
Cláudio Gomes, Aarhus University, Centre for Digital Twins
Hugo Daniel Macedo, Aarhus University, Centre for Digital Twins

Editors:

Casper Thule, Aarhus University, Centre for Digital Twins

Document History

Ver	Date	Author	Description
0.01	November 18, 2019	Casper Thule	Initial Version.

Abstract

TBD

Contents

1 Introduction

Maestro is a plugin-based framework built for specifying and executing co-simulations based on The Functional Mock-up Interface 2.0 and 2.1 standard for co-simulation. It is currently being developed.

There are three steps involved in conducting a co-simulation using Maestro:

1. Creating a specification of a FMI-based co-simulation to be executed.
2. Verifying the specification.
3. Executing the specification.

The specification of a co-simulation is based on a Domain Specific Language (DSL) called Maestro Base Language (MaBL). An MaBL specification is created by adding, editing and/or removing nodes in the Abstract Syntax Tree (AST) of the specification in question and not by writing it by hand. The main reason for this approach is the effort involved in developing Integrated Development Environment functionality for MaBL. Some of the major goals for this language are:

Goal 1: MaBL shall be capable of expressing both industry and research-related co-simulation scenarios.

Goal 2: MaBL shall be constrained such that specifications can be verified.

Goal 3: It shall be possible to serialise an MaBL specification such that it can be passed to a runtime for execution.

It is expected that aligning these goals will present challenges and for this reason, the initial version of MaBL does not cover all considered features and use cases. The considered features and use cases are available at TODO REF FEATURES AND USE CASES.

The process of creating a specification is to be carried out via plugins, which leads to another part of the Maestro framework. Maestro is based on plugins that offer certain functionality. For example, a plugin can generate the MaBL specification for initializing the FMUs of a co-simulation. Plugins can be combined such that the joined MaBL specification specifies a complete co-simulation that can be executed by a runtime.

The previous paragraph mentioned "a complete co-simulation", which has not been qualified. This leads us to the next part - verifying the specification. The verification of specifications, also based on plugins, will give meaning

to “a complete co-simulation”, which might be different from scenario to scenario.

The final part of conducting a co-simulation is to execute the verified specification. Maestro is envisioned to provide a C++-based runtime for executing MaBL specifications. However it is possible to provide other runtimes since a MaBL specification can be serialised. For the reason, creating, verifying and executing specifications are decoupled.

The features and use cases being considered to provide support for by Maestro is described in section 2. section 3 presents an extended description of how the different components of Maestro are joined to conduct a co-simulation. Next, section 4 presents the MaBL specification language. The publication then elaborates on plugins for creating and verifying specifications in section 5. Afterwards, section 6 presents examples and finally, section 7 presents ideas on the future work of Maestro.

2 Features and Use Cases

3 Approach to Conducting a Co-simulation

4 Maestro Base Language

5 Plugins

6 Examples

7 Future Work

8 Legacy Context

The Context contains information available to Maestro-Program, plugins and Maestro-Runtime. These entities can also add information to the Context by creating a new Context.

8.1 Context Structure

This section describes the structure of the Root Context and thereby what is supported natively. The reason for natively supporting some data is that it is considered essential for FMI co-simulation. First, the entries are presented with an overall description. Afterwards, the types mentioned in the overall description are described. Note, that not all entries are populated by Maestro-Program prior to employing plugins. Some of these entries will be populated by plugins. TODO: Clearly describe which entries Maestro-Program populates.

Overall Structure

RawFMUs The raw FMU information. Perhaps from a UI. Type: Map[FmuKey, FmuPath]

RawInstances The raw instance information. Perhaps from a UI. Type: Map[FmuKey, List[InstanceKey]]

RawConnections The raw connections information. Perhaps from a UI. Type: Map[FmuKey, FmuInstance, ScalarVariableName, Set[FmuKey, FmuInstance, ScalarVariableName]]

RawParameters The raw parameters information. Perhaps from a UI. Type: Map[ParameterKey, Value] and Map[FMU, Map[Instance, Map[ScalarVariableName, ParameterKey]]]

Parameters Parsed parameter information. Map[ScalarVariableID]

FMUsWithInstances Enriches RawFMUs with ModelDescription information and connects FMUs to their respective instances. Type: Map[Fmu, Set[Instance]]

Connections The connections based on FMUsWithInstances and variables from the corresponding ModelDescription files. Type: Set[Connection]

SortedDependantVariables An list that describes the order of setting and getting dependant scalar variables according to internal and external dependencies. Type: List[ScalarVariableID]

Custom This entry can be used freely. Type: Map[CustomDataKey, Any].

Types

FmuKey String: Unique identifier for a given FMU

FmuPath URI: Location of the FMU

InstanceKey String: Unique identifier for a given instance

ScalarVariableName String: Name of a given scalar variable. TODO: Perhaps valuereference, perhaps both?

Fmu Data Object with FmuKey, FmuPath, ModelDescription.

Instance Data Object that consists of parent FMU and InstanceKey

ModelDescription The parsed model description.

Connection Data Object with from of type ScalarVariableID and to of type Set[ScalarVariableID]

ScalarVariableID Data Object that consists of Instance and ScalarVariableValueReference

CustomDataKey String: Unique identifier for some custom data.

Any Represents the value associated with a CustomDataKey. TODO: Represent as Sum Type: Any (Any) | JSON (String) | Text (String) | Byte (Array[Byte]) | ?

9 Legacy Introduction

The framework is divided into two main parts: Maestro-Program and Maestro-Runtime. These entities and the flow of conducting a co-simulation using MaestroV2 is depicted in fig. 1 and described in this section.

Maestro-Program is the entity that controls the process of creating a program. A Program specifies how to conduct a co-simulation and consists of commands to be carried out by Maestro-Runtime. In order to create a Program, Maestro-Program employs plugins. The creation of a Program is referred to as the Program Phase.

Maestro-Runtime is the entity that controls the process of executing a program. The execution of a program is referred to as the Execution Phase.

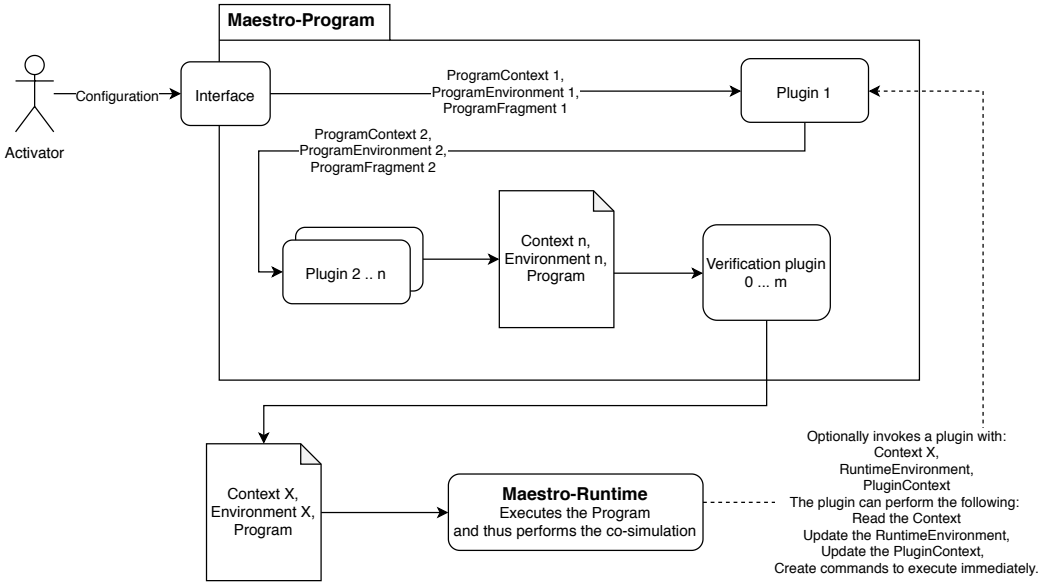

Figure 1: Conducting a co-simulation with Maestro-Program and Maestro-Runtime

The following paragraphs describes the content of the figure. Initially, some terminology and definitions are presented followed by a description of the consecutive behavior.

Context Context is data and information related to the co-simulation. For example, the FMUs to employ in a given co-simulation or the dependencies between the variables of the FMUs for a given co-simulation.

Environment The environment is information related to variables. For example, type information or the value of a given variable.

ProgramContext Terminology for the Context used in relation to Maestro-Program.

ProgramEnvironment Terminology for the Environment used in relation to Maestro-Program.

RuntimeContext Terminology for Context used in relation to Maestro-Runtime.

RuntimeEnvironment Terminology for the Environment used in relation to Maestro-Runtime. This contains i.e. variables in scope and values of variables.

PluginContext A plugin specific Context. Maestro-Runtime does not know its content and the code to process it must be provided by the respective plugin.

Configuration Configuration describing how to create the Program, i.e. which FMUs and plugins to use.

Root Context Terminology for the initial Program Context. Examples of data in the Root Context is i.e. FMUs to use in a co-simulation and parameters for the FMUs.

Program A Program is a complete in the sense that it can be passed to Maestro-Runtime for execution as opposed to a ProgramFragment described below.

Program Fragment A Program Fragment is part of a Program.

Plugin During the Program phase a plugin can read/update the Program Fragment, and/or read/update the ProgramContext and/or read/update the ProgramEnvironment. An example of ProgramContext information that a plugin can add is the dependencies between the variables of the FMUs. An example of a Program Fragment that a plugin can create is the necessary commands to perform initialisation of the FMUs. During the Runtime phase a plugin can read/update the RuntimeEnvironment, read/update the PluginContext and/or create commands to be executed immediately.

The Activator in fig. 1 is the entity (person or tool) that launches a co-simulation. The Activator shall provide a Configuration. See TODO.

Maestro-Program invokes the plugins according to the configuration. This is demonstrated in fig. 1 where Plugin 1 receives ProgramContext 1, ProgramEnvironment 1, ProgramFragment 1 and creates: ProgramContext 2, ProgramEnvironment 2 and ProgramFragment 2. This is then passed to Plugin 2 and so on until no more plugins are specified. At this stage it is expected that a Program has been created. It is then possible to verify the Program, which is also based on plugins. The verification plugins can report their results but cannot update the Program, context or environment. The result of the Program phase is: A Program (Program in the figure), a Context (Context X in the figure) and an Environment (Environment X in the figure)

Maestro-Runtime executes the Program and can utilise the related Context and Environment. It is possibly to create commands in the Program that prompts Maestro-Runtime to invoke a specific plugin. The plugin will be invoked with the initial Context (Context X in the figure), the Runtime Environment (RunTimeEnvironment in the figure) and a Context for the specific plugin (PluginContext in the figure).

9.1 TO BE DONE

References

A List of Acronyms

DSL Domain Specific Language