CoHLA User Manual

Thomas Nägele
t.nagele@cs.ru.nl

25th June 2019

Contents

1 Introduction

This manual describes the basics on how to use CoHLA (Configuring HLA). The project aims for an easier implementation of co-simulations of systems using the High Level Architecture. Using a DSL, one can easily describe a model-based co-simulation. From this co-simulation definition, the framework will generate code that can be compiled and simulated using OpenRTI.

2 Installation

Is is assumed that CoHLA has already been installed on the system, as well as all its dependencies. If either one of the components is not installed, please make sure to do so according to the Installation Manual.

3 Definitions

The manual sometimes refers to some executable, workspace or location. To avoid having to repeat the definition of some of these in each section, we will introduce them below.

CoHLA Workspace This is the workspace where the CoHLA projects will be created and maintained.

Project Directory This is the directory in the CoHLA Workspace where the project is located. This directory may contain the models (possibly in a sub-directory) and all CoHLA source files.

Source Generation Directory This is the directory in the Project Directory where the sources generated by the CoHLA source generator will be created. These sources are typically located in the *src-gen* directory in the Project Directory.

Model Directory This may be a directory in the Project Directory in which all models for the co-simulation are stored. The Model Directory may also be equal to the Project Directory. Is is **not** recommended to store the models in the Source Generation Directory.

Library Directory The directory in which the OpenRTI libraries are installed. By default, these libraries are installed in `C:\opt\OpenRTI-libs` (for Windows) or `\opt\OpenRTI-libs` (for Linux/Mac).

4 CoHLA project definition

This section outlines all components that can be used to specify a co-simulation of models using CoHLA. The section layout of this section matches the order in which the components should be defined.

4.1 Imports

CoHLA allows the use of imported CoHLA files. This can be particularly useful when specifying Federate Classes in different source files. An import consists of the keyword `import` followed by the file name to import as a string. Zero or more imports may be specified in a file.

```
1 import "model.cohla"
```

4.2 Environment (required)

The environment is used to specify some code generation settings.

4.2.1 HLA version (required)

Specifies the HLA RTI implementation that is used. Currently, the generator is only capable of generating code for OpenRTI. The version is specified by the starting a configuration block with the keyword `RTI`. The block starts with the RTI implementation to generate code for, which is one of the following RTIs.

`OpenRTI` Generates C++ code for OpenRTI[1].

`PitchRTI` Generates C++ or Java code for Pitch [2].

`Portico` Generates Java code for Portico[3].

The file path to the libraries for this RTI should also be specified by using the `Libraries` keyword, followed by a string containing the path. Dependencies may be located on a different location on the file system. This location may be provided by using the optional `Dependencies` keyword.

```
RTI {
  OpenRTI
  Libraries "/opt/orti-libs"
  Dependencies "/opt/deps"    // Optional
}
```

4.2.2 Source directory

Overrides the default output directory for the sources that are being generated. The directory is set to *src* by default.

```
SourceDirectory "sources"
```

4.2.3 Print level

Sets the print level of each of the generated federates. By default, on every advancement in time, the state of the federate (its attribute values) are being printed to the console. Other options are the following.

`State` Prints all its attribute values upon time advancement. (default)

`Time` Prints only the time upon time advancement.

`None` Disable printing.

```
PrintLevel None
```

4.2.4 Publish only changes

When the keyword `PublishOnlyChanges` is specified, all federates only publish attributes after time advancement when one or more of its published attributes has changed.

```
Environment {
  RTI {
    OpenRTI
    Libraries "/opt/OpenRTI-libs"
  }
  PublishOnlyChanges
}
```

Listing 1: A sample Environment definition.

[1] https://sourceforge.net/projects/openrti/
[2] http://www.pitchtechnologies.com/products/prti/
[3] http://www.porticoproject.org/

4.3 Federate classes (required)

A Federate Class represents a simulation model that could be connected to the co-simulation. Multiple instances of the same Federate Class can be connected to the co-simulation. The Federate Class is used to specify some default parameters and its attributes. Every Federate Class is identified by a name.

```
1 FederateClass SimModel { ... }
```

4.3.1 Type (required)

To specify the type to use for the model, a `Type` must be specified. Values can be one of the following.

- **POOSL**: POOSL model using the Rotalumis simulator.
- **FMU**: FMI model to be controlled using the FMI=library.
- **CSV-logger**: Logger federate exporting a CSV file.
- **BulletCollision**: Collision detection using the Bullet Physics Library.
- **None**: No simulator specified.

When None is specified, no wrapper code for a simulator is generated. A sample definition is the following.

```
1 SimulatorType FMU
```

4.3.2 Type configuration

Some simulator types require some additional configuration to work. For now, this is only the case for POOSL and CSV-logger federates. The configuration is specified in a block, directly following the simulator type.

POOSL configuration

Processes are used to specify a comprehensive mapping to attributes in a POOSL model. Since a POOSL model executes as a set of processes, a path to the process in which an attribute is defined is required to read the attribute from the simulator. The identifiers of these processes are then used when defining attributes for the Federate Class. All processes should be specified within a `Processes` block.

```
1 Processes { ... }
```

Every process consists of a name (ID) and a path (string) , separated by the keyword `in`: `{ID} in {STRING}`.

```
1 Processes {
2   process1 in "path/to/process"
3   process2 in "path/to/second/process"
4 }
```

CSV-logger configuration

The optional configuration for a CSV-logger type of federate only consist of a default measure time to record the simulation. This value is provided after the `DefaultMeasureTime` keyword.

```
1 DefaultMeasureTime 5800.0
```

4.3.3 Published interactions

Currently not in use.

4.3.4 Subscribed interactions

Currently not in use.

4.3.5 Attributes

Attributes are variables that may be shared with other simulators or act as inputs from other simulators (or both). Basic attributes can be defined within a single line, although some more HLA-specific configuration can be done in an option block. Attributes must be defined in an `Attributes` block.

```
Attributes { ... }
```

A basic attribute is defined by the following:
`{SharingType} {DataType} [Collision]? [{MI_Operator}]? {ID} {Alias}?`
The `Collision` keyword sets the attribute to be the receiver of collider state for the collision simulator. When the DataType is Boolean, this is either `true` or `false`; when it is an Integer, the number of collisions found is stored.

SharingType

The sharing type specifies whether the attribute is an input or an output attribute. The attribute can also be both input and output. Possible values are `Input`, `Output`, `InOutput`.

DataType

The data type specifies the data type the attribute stores. It can either be `Integer`, `Long`, `Real`, `Boolean` or `String`, all corresponding to their C++ or Java native types.

MI_Operator

The MultiInput operator specifies how to determine the attribute value when multiple outputs are connected to it. By default, the latest value that is received is stored. Possible values are `None` (default), `&&`, `||`, `*` and `+`.

Alias

The alias specifies how the corresponding attribute value is known in the simulator. For Rotalumis simulators, this requires both a process identifier and a name.

```
in process1 as "difficultName"
```

For FMU simulators, it only requires a name.

```
as "difficultName"
```

A possible definitions of attributes are the following.

```
Attributes {
  InOutput Real power
  Input Real actorXPosition as "x_position"
  Output Boolean [Collision] hasCollisions
  InOutput Boolean [||] activity in c as "activity"
}
```

4.3.6 Parameters

Parameters are basically attributes that can be set during initialisation. These are defined by the following:
`{DataType} {ID} "{Alias}" (in [Process])?` Again, the process property should only be specified when the federate class is based on a POOSL model. The Alias property is a string that specifies the attribute name in the model. The DataType property is equal to the one in the Attribute definition as described above. Parameters are wrapped in an “Parameters” block. Samples are listed below.

```
Parameters {
  Real holdTime "HoldTime"
  Real "HoldLevel"
  Boolean active "Active"
  Real VacantLevel "VacantLevel" in c
}
```

4.3.7 Purpose

Currently not in use.

4.3.8 Time policy

Sets the HLA time policy for this Federate Class. Value can be `RegulatedAndConstrained`, `Regulated`, `Constrained` or `None`. These values correspond with the HLA federate time policy.

```
1 TimePolicy RegulatedAndConstrained
```

4.3.9 Default model

To specify the default model – or default models – to use in the simulation, this can be specified using the `DefaultModel` property. This property should be used for both FMU and Rotalumis simulator types. When it is used for BulletCollision simulator types, multiple models may be given by providing multiple file paths as strings. Two samples are the following.

```
1 DefaultModel "../path/to/model/Model.fmu"
2 DefaultModel "models/model1.json" "models/model2.json" "models/model3.json"
```

4.3.10 Advance type

`AdvanceType` is a property to provide a default step method to the federate. Options are the following.

- **TimeAdvanceRequest**: Request time advance grants to RTI.
- **NextMessageRequest**: Request next message requests to the RTI.

A sample definition is the following.

```
1 AdvanceType NextMessageRequest
```

4.3.11 Default step size

`DefaultStepSize` can be used to specify the default step size to take for this simulator during the simulation. The keyword is followed by a double value that indicates the step size. For Message-based steps, this value represents the time-out to wait for a message. For POOSL models, this property does not have any effect, because these models provide their step size to the simulator. A sample is the following.

```
1 DefaultStepSize 3.0
```

4.3.12 Default lookahead

The default lookahead value can also be specified using the `DefaultLookahead` property. The property is very similar to the step size and a sample is displayed below.

```
1 DefaultLookahead 0.1
```

4.3.13 Simulation weight

The simulation weight represents the computational weight of the model simulation. A very computation intensive simulation should be assigned a higher weight than a very light simulation model. This value is used to create a distribution across multiple nodes automatically. The default value for each simulation is 1.

```
1 SimulationWeight 3
```

4.4 Interfaces

Interfaces can be created for federate classes that typically have many instances being connected to each other by equal attribute connections. They specify the attribute connections on a class level instead of instance level, after which two instances can be connected to each other without having to specify every attribute connection between them again. An interface starts with the `Interface` keyword, after which both federate classes are specified that use this interface to be connected to each other. Then a block is started in which all attribute connections are specified. The layout of an Interface specification is as follows:
`Interface between [FedClass] and [FedClass] { {Connection}+ }`

4.4.1 Interface connection

Each of the interface connections represents the connection on an attribute level: which attribute of one federate is connected to which attribute of the other federate. The format for this is the following:
`{ [FedClass].[attr] <- [FedClass].[attr] }`
Both attributes should be a member of the corresponding federate class. The first attribute receives its input from the second attribute. A sample interface is displayed below.

```
1 Interface between FedClassA and FedClassB {
2   { FedClassA.voltage <- FedClassB.outp }
3   { FedClassB.inp <- FedClassA.encoder }
4 }
```

4.5 Configurations

A `Configuration` is a block definition containing a initialisation configuration for a federate class. It is defined like the following:
`Configuration {ID} for [FederateClass] { {InitAssignment}+ }`
The ID gives the configuration a name and the FederateClass specifies for which class the configuration is.

4.5.1 Initialisation attributes

Every `InitAssignment` assigns a value to a `Parameter` of the federate class. All values are put in parenthesis as string notation, regardless of the data type. An assignment looks like the following.
`[Parameter] = "{Value}"`
A sample Configuration for the parameters that were given previously is the following.

```
1 Configuration Large for Room {
2   holdTime = "300.0"
3   holdLevel = "50.0"
4   active = "true"
5   VacantLevel = "10.0"
6 }
```

4.6 Interactions

Currently not in use.

4.7 Federations

Federations are used to specify a co-simulation. In the basis, it consists of a number of simulations (instances) and connections between them. A couple of extensions are added to provide some additional features. A `Federation` looks like the following:
`Federation {ID} { {Instances}? {Connections}? ... }`

4.7.1 Instances

A Federation consists of a number of simulations that are connected to each other. Each of these simulations is an `Instance` of a federate class as specified earlier. An instances block looks like the following:
`Instances { ({ID} : [FederateClass])+ }` A sample instances block is the following.

```
Instances {
  sim1 : FedA
  sim2 : FedA
  sim3 : FedB
}
```

4.7.2 Connections

Connections are similar to the interface connections. Regular connections, however, connect attributes on an instance level. A connections block looks like the following:
`Connections { {Connection}+ }` A connection can either be an attribute connection, an instance connection or a logger connection. An attribute connection connects two attributes from two instances to each other. Such a connection looks like the following:
`{ [FedClass].[Attribute] <- [FedClass].[Attribute] }`
An instance connection connects multiple attributes at once according to a previously defined `Interface`. Which interface to use should not be specified. An instance connection looks like the following:
`{ [Instance] - [Instance] }`
A logger connection connects multiple output attributes from different instances to a single logger instance. Such a connection looks like the following:
`{ [Instance] <- [Instance].[Attribute] (, [Instance].[Attribute])* }` Altogether, a sample connections block could be the following.

```
Connections {
  { sim1.inp <- sim3.outp }          // Attribute
  { sim2.inp <- sim3.outp }          // Attribute
  { sim1 - sim2 }                    // Instance
  { logger <- sim1.outp, sim2.outp } // Logger
}
```

4.7.3 Situations

A `Situation` can be used to specify an initialisation configuration for a Confederation. A Situation consists of a set of initialisation attribute assignments and/or applications of configurations to federate instances. Situations can extend each other to support several layers of initialisation configurations to be stacked. A Situation looks like the following:
`Situation {ID} (extends [Situation])? { {SituationElement}+ }`
Every SituationElement is either an attribute initialisation or an application of a model configuration.

Initialisation attribute initialiser

The initialisation attribute initialiser specifies the value for one of the initialisation attributes for a specific federate instance in the Confederation. This SituationElement looks like the following:
`Init [Instance].[InitAttribute] as "{Value}"`
The Instance should already be defined in the Confederation and only initialisation attributes can be assigned. The value must be specified as a String.

Application of model configuration

An application of a model configuration applies a model configuration to a specific federate instance. This is particularly useful when assigning multiple initialisation attributes at once. This SituationElement looks like the following:
`Apply [ModelConfiguration] to [Instance]`
When extending Situations, higher-level Situations might overwrite initialisation attribute values specified in lower-level Situations. A sample Situations block might look like the following.

```
Situation sit1 {
  Apply large to room1
```

```
  Init room1.heaterSize as "3.2"
  Apply small to room2
  Init room2.heaterSize as "1.1"
}
Situation sit2 extends sit1 {
  Apply medium to room1
  Init room1.heaterSize as "2.3"
}
```

4.7.4 Scenarios

A `Scenario` is a predefined sequence of events taking place during the co-simulation. A scenario optionally starts with an end time specified for when the simulation should be stopped:
`AutoStop: ({FLOAT} | no)` Every `Scenario` has an ID and supports both assignments of attributes in the co-simulation and messages being sent over sockets. For the latter case, sockets should be specified first in the `Scenario` block. The targeted instance connects to the specified socket after starting the co-simulation. A socket specification looks like the following:
`Socket {ID} for [Instance] on "{Hostname}":{Port}`
After the (optional) specification of the sockets, the sequence of events is specified. Every `Event` occurs at a given time and is either an assignment or a socket message (in bytes). An assign event looks like the following:
`On {Time} Assign "{Value}" to [Instance].[Attribute]`
A socket event looks like the following:
`On {Time} Send [Socket] bytes {Byte}+ (, {Byte}+)?`
Here, the Socket refers to one of the previously defined sockets and Byte represents a sequence of bytes in hexadecimal representation, such as `0x01 0xe7 0xB1`. A sample Scenarios block is the following.

```
Scenario sampleScenario {
  AutoStop: 30.0
  Socket sock1 for room1 on "localhost":10001
  Socket sock2 for room2 on "localhost":10002
  On 10.0 Send sock1 bytes 0x01 0x02 0x03
  On 10.0 Send sock2 bytes x01 x02 x03          // Byte notation is allowed too
  On 15.0 Assign "false" to room1.heaterState
}
```

4.7.5 Fault scenarios

A federation might contain a number of FaultScenarios. Every `FaultScenario` looks like the following:
`FaultScenario { {Fault}+ }`
For most of the faults, either a single moment in time, the period starting from a specified time, or a time range can be specified. This definition is called a `Range` and looks like the following:
`(On {Time}) | (From {Time} (to {Time})?)`
There are different types of Faults that could occur; these are listed below.

- `AccuracyFault`: An AccuracyFault introduces some variance on an attribute. It looks like the following:
  `Variance for [Instance].[Attribute] = {Float}`
- `TimedAbsoluteFault`: A TimedAbsoluteFault sets the value of an attribute to a predefined value at a specific time. It looks like the following:
  `{Range} set [Instance].[Attribute] = "{Value}"`
- `TimedConnectionFault`: A TimedConnectionFault interrupts the connection of an attribute to the other attributes, disabling it to publish its most recent values. It looks like the following:
  `{Range} disconnect [Instance].[Attribute]`
- `TimedOffetFault`: A TimedOffsertFault introduces an offset to an attribute value; it looks like the following:
  `{Range} offset [Instance].[Attribute] = "{Value}"`

All `Value` elements in parenthesis are String representations of the attribute value to be assigned.

```
FaultScenario sampleFaultScenario {
  Variance for fed1.attr2 = 0.3
  From 2.0 to 5.0 set fed2.attr1 = "2.5"
  From 12.0 offset fed4.attr3 = +0.2
  On 8.0 disconnect fed3.attr1
}
```

4.7.6 Design Space Exploration

Every `DSE` has an ID and some basic settings: `SweepMode`, `Scenario` and `FaultScenario`. These are all optional. The `Scenario` and `FaultScenario` properties specify which of these components to run for each of the design space configurations when co-simulating. The SweepMode specifies how the configurations should be sweeped, either `Independent` (default) or `Linked`. Then, the design space is specified by providing Situations, Configurations and Initialisation Attributes to sweep. Each of these lists is specified as listed below.

- `Situations: [Situation] (, [Situation])*`
- `Configurations for [Instance] : [Configuration] (, [Configuration])*`
- `Set [Instance].[InitAttribute]: {Value} (, {Value})*`

A Sample DSEs block looks like the following.

```
DSE dse1 {
  SweepMode Independent
  Scenario sampleScenario
  Faults sampleFaultScenario
  Situations: sit1, sit2
  Configurations for room1: small, medium, large
  Configurations for room2: small, medium, large
  Set room1.heaterSize: 1.2, 2.3, 3.4
  Set room2.heaterSize: 1.1, 2.2
}
```

The sample DSE "dse1" has a design space of 108 configurations.

4.7.7 Metric sets

A federation may consist zero or more `MetricSet`s. Every `MetricSet` has an ID and requires a MeasureTime to be defined. This looks like the following:
`MetricSet {ID} { MeasureTime: {Float} {Metric}+ }`
Every Metric also has an ID and can either be one of the following metrics.

- `Absolute? EndValue [Instance].[Attribute] (relative to [Instance].[Attribute])?`
- `Squared? Error [Instance].[Attribute] (relative to [Instance].[Attribute])?`
- `Timer for [Instance].[Attribute] {Comparator} {Value} (EndCondition (after {Time})?)?`
- `Minimum of [Instance].[Attribute]`
- `Maximum of [Instance].[Attribute]`

`Absolute` and `Squared` are optional extensions to the `EndValue` and `Error` metrics respectively, as well as the `relative to` property, which allows the user to request a metric value relative to another attribute value. The `Comparator` can be one of the following: `<, <=, ==, >=, >, !=`. Once the condition made by comparing the attribute value to the specified value using the comparator yields `true`, the time is set a metric result for the timer. Optionally, the simulation may be trigger to exit when this timer yields `true`. A delay before closing can also be specified. Minimum and maximum value are self-explaining.

4.7.8 Distributions

`Distribution` configurations for use of distributed simulation over multiple nodes may be specified for a federation. Every `Distribution` has an ID and a number of nodes specified. This looks like the following:
`Distribution {ID} over {Number} systems { {SystemSet}+ }`
Every `SystemSet` is assigned a identification number of the node and a list of federate instances that should be executed on the node. This looks like the following:
`System {Number}: [Instance]+`
A sample of a `DistributionSets` block is the following.

```
1 Distribution dist1 over 2 systems {
2   System 0: fed1 fed2 fed3
3   System 1: fed4 fed5 fed6
4 }
5 Distribution dist2 over 3 systems {
6   System 0: fed1 fed4
7   System 1: fed2 fed3 fed5
8   System 2: fed6
9 }
```

5 Running a co-simulation

Once a Confederation has been defined, the CMake project, configuration files and run-script are generated in the Source Generation Directory.

5.1 Configuration files

The `conf` directory contains all configuration files divided in a couple of subdirectories. Each of these directories and their contents are listed below. For each FederateClass having Model Configurations specified, a new directory is created.

- `conf`
  - Topology file holding information on instances and connections (.topo)
- `conf/Confederation`
  - Situation configurations (.situation)
  - Scenarios (.scenario)
  - Fault scenarios (.fscenario)
  - Design Space Exploration configurations (.dse)
  - Metric sets (.metrics)
  - Distributions (.distribution)
- `conf/FederateClass`
  - Model configurations (.init)

5.2 Run-script

In the root of the Source Generation Directory, a Python run-script (run.py) is generated. This run-script can be used to build and run the co-simulation. The script must be executed from the command line and supports a set of flags and properties to specify its actions. All options are listed below and are categorised.

5.2.1 Basic behaviour

These flags tell the script what to do.

- `-b` Build the generated sources using CMake.
- `-d` Display the active configuration.
- `-e` Start the simulation using the active configuration.
- `-h` Display the help message.

5.2.2 Configurables

These properties can be used to configure the co-simulation.

- `-t` x, `--topology` x
  Sets the topology configuration file to x.
- `-f` x, `--fom` x
  Sets the FOM XML file to x.
- `-r` x, `--rti` x
  Sets the RTI address (hostname) to x.
- `-s` x, `--situation` x
  Sets the situation configuration file to x.
- `-u` x, `--scenario` x
  Sets the user scenario file to x.
- `-m` x, `--metrics` x
  Sets the configuration files for metrics to x.
- `--faults` x
  Sets the fault scenario file to x.
- `--dse` x
  Sets the Design Space Exploration configuration file to x.

5.2.3 Distribution setup

These options can be used to start a distributed co-simulation.

- `--distribution` x
  Set the predefined distribution configuration file to x.
- `--systemId` x
  Set the node/system ID of the current node to x.
- `--nrOfSystems` x
  Set the total number of nodes in the distribution execution to x.
- `--parentHost` x
  Set the parent hostname to x.
- `--parentPort` x
  Set the parent port to x.

5.2.4 Other options

A couple of other options to apply changes are available.

- `-v`, `--verbose`
  If set, all outputs from external commands – such as CMake – will be displayed in the console.
- `--disable` x
  Disable the execution of federate x.
- `--cmake=`x
  Add additional flags x to the CMake build command.
- `--render`
  If set, collision simulators will also render the provided models.

5.2.5 Federate-specific options

All federates have their own additional options. Examples are the measure time, step size, lookahead en models to use for the co-simulation. Also, initialisation attributes can be overridden by providing values to the run-script. All additional options can be seen when a topology is set and the `-h` option is provided. An example to request all options from the run-script is the following.

```
./run.py -t conf/sampleFederation.topo -h
```

Other useful commands are the following.

```
// Build all sources and display output
./run.py -bv

// Build and start the basic configuration
./run.py -t conf/sampleFederation.topo -be

// Set situation and fault scenario, build and start
./run.py -t conf/sampleFederation.topo -u conf/SampleFederation/WorkDay.scenario -s
    conf/SampleFederation/Medium.situation --faults conf/SampleFederation/errors.
    fscenario -be
```