Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Updated Integration of Simulators in the INTO-CPS Platform

Deliverable Number: D4.2b

Version: 1.0

Date: December 2016

Public Document

http://into-cps.au.dk

Contributors:

Adrian Pop, LIU
Victor Bandur, AU
Kenneth Lausdahl, AU
Marcel Groothuis, CLP
Tom Bokhove, CLP

Editors:

Frank Groen, CLP

Reviewers:

Carl Gamble, UNEW
Ana Cavalcanti, UY
Alie El-Din Mady, UTRC

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	13-05-2016	Marcel Groothuis	Initial document version
0.2	06-07-2016	Marcel Groothuis	Allocation of responsibilities and provisional dates for inclusion
0.3	14-10-2016	Frank Groen	General changes and addition of Stability section
0.4	31-10-2016	Frank Groen	Some modifications
0.5	31-10-2016	Kenneth Lausdahl	Added Overture section
0.6	31-10-2016	Adrian Pop	Added OpenModelica section
0.7	01-11-2016	Frank Groen, Kenneth Lausdahl	Last modifications before internal review
1.0	15-12-2016	Frank Groen, Tom Bokhove	Review comments addressed.

Abstract

This deliverable contains the design specifications for integration of simulators (OpenModelica, Overture and 20-sim) with the INTO-CPS co-simulation orchestration engine (COE) at the end of the second year of the project. The integration of the simulation tools into the COE will use Functional Mockup Interface (FMI) with INTO-CPS extensions. Monitoring stability of the co-simulation is discussed in the chapter on the COE.

Contents

1 Introduction

This deliverable contains the design for the integration of the simulators OpenModelica, Overture and 20-sim with the co-simulation orchestration engine (COE) using FMI2. For monitoring the overall stability of a Co-Simulation it may be that extensions to the FMI standard are necessary. This deliverable is a continuation of Deliverable D4.1b [PBLG15] and it describes the updates that have been developed in the second year.

The integrated simulators in this project are:

- OpenModelica [Fri04], `https://openmodelica.org`
- Overture [LBF+10], `http://overturetool.org/`
- 20-sim [KGD16], `http://www.20sim.com/`

1.1 Requirements

The high-level requirements from the INTO-CPS requirements report in deliverable D7.3 [LPH+15] with focus on the INTO-CPS Integration of Simulators are presented below for the different baseline tools.

- Requirement **0007** - The COE must be able to monitor the overall stability of a co-simulation based on the suggested step-size, step-size tolerance as well as input values (min, max and nominal) of FMUs.
- Requirement **0008** - The COE must have algorithms in place to increase the overall stability of a co-simulation.
- Requirement **0009** - The OpenModelica tool must provide an INTO-CPS FMI tool wrapper that is compliant with the COE.
- Requirement **0010** - The 20-sim tool must provide an INTO-CPS FMI tool wrapper that is compliant with the COE.
- Requirement **0011** - The Overture tool must provide an INTO-CPS FMI tool wrapper that is compliant with the COE.

1.2 Related Work

Several approaches have been proposed in the past dealing with integration of simulators at different levels:

- simulator-tool level - the tools are called as slaves by a master (covered by FMI for Model Exchange);
- model-export level - the tool can export a model that can be imported in another tool (covered by FMI for Co-Simulation);
- source-code level - the tool can export source code that can be integrated with source code exported from other tools.

At the *simulator-tool level* several tools (Adams, Modelica, etc) were integrated using co-simulation within the SKF BEAST tool [SNF05], [Sie10]. Also integration of Overture and 20-sim has been achieved [GMF12] before in the DESTECS [DES09], [LRV+11] EU project at the *simulator tool level*.

At the *model-export level*, for example, 20-sim can export Matlab S functions. [KGD16]

At the *source-code level*, 20-sim is able to generate C-code based on a template that enables it to embed the code in a bigger framework. Please see deliverable D5.1d – *Design Principles for Code Generators* [HLG+15] for related work into integration at the *source code level*.

In this project the integration is performed at *model-export level* where models are exported from tools as FMUs for co-simulation based on the FMI2 standard. The exported FMUs can then be co-simulated using the COE.

The following research articles might proof useful for the issues described in this deliverable, and might be further investigated in the scope of year three of the INTO-CPS project: [Vie14], [And16], [SP16] and [SBE+14].

2 Integration of Simulators

Integration of simulators in the INTO-CPS COE is achieved via the FMI 2.0 standard, namely FMI 2.0 for co-simulation. Each of the simulators has implemented support for the FMI 2.0 standard.

In the next sections we give the details of each simulator and its support for the FMI 2.0 for co-simulation integration.

2.1 Overture

Overture [LBF+10] is an Eclipse[1]-based open-source platform for the development and validation of VDM specifications. Because it is written in Java, it can run in any Java-enabled operating environment. Overture supports three dialects of VDM, namely the core VDM-SL specification language, the object-oriented VDM++, and its extension for real-time embedded systems, VDM-RT. The dialect of VDM relevant to the INTO-CPS project is VDM-RT. The key feature of Overture relevant to its integration with the INTO-CPS tool chain is its compliance with the FMI standard, which has been developed for the INTO-CPS project.

2.1.1 FMI Support

Overture has tool support for both import and export of FMI:

Import An FMI description can be imported into a VDM-RT model. When imported and no `HardwareInteface` is available, then it is created. Otherwise, it is updated with the difference between the model description and what is already defined.

Export FMU export is supported in two versions:

Tool Wrapper This exports the VDM sources together with a standalone version of Overture which provides an FMI interface. It includes all tracing and logging features available to Overture. This is implemented such that the VDM Interpreter and its FMI interface is included in the exported FMU. The supported platforms are Win32, Win64, Linux64, Darwin64 and require Java 1.8 to be installed and available in the PATH environment variable.

C Source Code This converts the VDM sources into a C implementation and exports this together with a small FMI wrapper that enables all periodic threads to be executed according to the doStep time period during a co-simulation.

The Tool wrapper has one limitation concerning the FMI 2.0 standard. It does not support the *Initialization Mode* described in the FMI standard under Section 4.1.2. Initialization Mode is not supported since the VDM interpreter is unable to do any calculation of output variables unless a sufficient time step is given to `doStep`.

[1] `www.eclipse.org`.

Table 1: FMI functions currently not implemented

FMI 1.0	**FMI 2.0**
Not supported	reset getFMUstate setFMUstate freeFMUstate serializedFMUstateSize serializeFMUstatev deSerializeFMUstate getDirectionalDerivative setRealInputDerivatives getRealOutputDerivatives cancelStep *getMaxStepsize*

The VDM language is focused on modelling discrete systems and therefore does not include the notion of derivatives. It thus does not support the functions related to derivatives, as listed in Table 1. The VDM language includes concurrency and as a result of its implementation in the VDM interpreter [LLB11], where Java threads are used, the functions to get and set state are also not supported.

In its current state, the FMI export feature supports getting and setting Reals, Integers, Booleans and String, as well as the INTO-CPS specific `getMaxStepsize` function, which returns the next smallest step-size where the VDM model changes its outputs. This method is what enabled variable stepping without the need for get-, set-state. The export does not provide a way to define units, since these cannot be explicitly modelled in the VDM language.

2.1.2 Mapping between VDM and FMI

Overture has been extended with FMI support through a dedicated FMI framework, a few modelling guidelines and tool support. The FMI interface is modelled as a class hierarchy of `Ports` matching the types from the FMI standard (`IntPort`, `BoolPort`, `RealPort`, and `StringPort`). These ports must then be declared in a special `HardwareInterface` class which represents the model variables part of the FMI model description. An in-

stance of this hardware interface must be available as a static instance variable in the system class.

The first guideline is that the system must contain the hardware interface. This is required because the extension to the Overture interpreter that handles the FMI embedding needs to be able to uniquely identify the instance holding the ports used to map the scalar variables from the FMI model description. An example of this is shown in Listing 1.

```
system System

instance variables

-- Hardware interface variable required by FMU Import/Export
public static hwi: HardwareInterface := new HardwareInterface();
 ...
end System
```

Listing 1: The Hardware interface variable definition in the system

The second guideline is that all port instances must be declared in the `HardwareInterface` class. To be recognized by the tools, a comment must be placed directly above the definition of the port, as shown in Listing 2.

```
-- @ interface: type = <type> (, name=<name>)?;
public <definition>
```

Listing 2: The FMI port declaration annotation

The `<type>` can either be `parameter` (only for values) or `input` / `output` for instance variables. The `<name>` is optional, but if defined, the name will be exported as in the model description.

To illustrate how such a mapping can be made, the Watertank example from the Crescendo tool [IPG+12] is used. The Watertank is a simple model of a controller that adjusts the level in a tank with a constant in-flow. It keeps the water level between a low and high mark by opening and closing a valve. The system model consists of a level sensor, valve actuator, the controller, a configuration of the system topology using buses and CPUs as shown in Listing 3, and a hardware interface shown in Listing 4. The system constructor, shown in Listing 3, shows how the sensor and actuator are instantiated with two different ports from the `hwi` instance. The ports can be passed as arguments allowing reused of e.g. a standard implementation of a Propor-

tional–Integral–Derivative controller by instantiating each instance with the ports required for sensor values and actuators. The `controller` is then deployed onto `cpu1` which uses the `CPU` configuration to determine execution time of expressions and statements [LLB11]. The ports are defined in Listing 4, where values are defined as parameters and instance variables are configured as either input or output. The rest of the model has been left out but is available as a case study named *Single Watertank* [PGP+16].

```
system Sys

instance variables

  -- Hardware interface variable required by FMU Import/Export
  public static hwi: HardwareInterface := new HardwareInterface();
  public static controller : [Controller] := nil;
  cpu1 : CPU := new CPU(<FP>, 5E3);

operations

public Sys : () ==> Sys
Sys () ==
  let levelSensor = new LevelSensor(hwi.level),
      valveActuator = new ValveActuator(hwi.valveState )
  in (
    controller := new Controller(levelSensor, valveActuator);
    cpu1.deploy(controller,"Controller");
  );
...
end System
```

Listing 3: The Watertank system

```
-- Generated class
class HardwareInterface

values
  -- @ interface: type = parameter, name="minlevel";
  public minlevel : RealPort = new RealPort(1.0);
  -- @ interface: type = parameter, name="maxlevel";
  public maxlevel : RealPort = new RealPort(2.0);

instance variables
  -- @ interface: type = input, name="level";
  public level : RealPort := new RealPort(0.0);
```

```
instance variables
  -- @ interface: type = output, name="valve";
  public valveState : BoolPort := new BoolPort(false);

end HardwareInterface
```

Listing 4: The Hardware interface class

2.1.3 Step-size Calculation

The VDM interpreter internally uses a variable step-size. The step-size is defined as the time it takes for the interpreter to execute expressions and statements between read of inputs and write to outputs. The interpreter will upon each call to `doStep` execute until it needs to read from an input or write to an output. If the time it took to perform the calculation is less than the time + step-size, then it will return `Ok` from `doStep`, and hold back any writes until the exact time where the write should occur is requested by do step. If the VDM interpreter is waiting for such a write then a subsequent call to `doStep` must have a step-size which is smaller or precisely matches the time when the write must occur. If not, then `Discard` is returned from `doStep`. The step is discarded to ensure that the `Ports` representing the FMI scalar variables always return the newest (current time) value when their values are read. If this is not guaranteed then it will eventually lead to invalid co-simulations, since the VDM model would not be able to detect what time a value read from a `Port` belongs to. This situation can lead to co-simulation results indicating an instability due to late control actions, when in fact the apparent instability is only an artifact of the mismatch between step-size and the discrete times when the VDM controller can read and write port values.

The VDM interpreter supports two ways to enable the COE to obtain or predict the step-size it will accept in a call to `doStep`. If the interpreter returns `Discard` then the function `getRealStatus` with `LastSuccessfulTime` can be used to obtain the time the interpreter will accept, or the INTO-CPS extension function `getMaxStepsize` can be used before taking a step to calculate the largest step the interpreter will accept, avoiding the need for roll-back in other FMUs, which is required to recover from a discarded step.

2.2 20-sim

2.2.1 Introduction

20-sim is a modeling and simulation program for mechatronic systems and control engineering on the Windows operating system. With 20-sim, multi-domain dynamic models can be analyzed in the time and frequency domain for modeling and control purposes. For rapid prototyping and HIL-simulation purposes, C-code generation support is available using C-code templates for various C-code objectives.

With respect to simulation, 20-sim supports continuous time simulations, discrete time simulations and hybrid simulations. For variable step-size integration method support, simulation back-stepping is available, but not for discrete time systems. External interfacing to 20-sim is available using scripting (XML-RPC), custom DLL functionality and CSV file variables.

2.2.2 FMI Support

20-sim has tool support for both import and export of FMI:

Import FMU import is supported in the following ways:

Interface Definition An FMI `modelDescription.xml` can be imported into 20-sim. This results in an empty 20-sim block with the corresponding FMI interface. After adding the wanted model implementation, it can be exported again as an FMU.

Co-simulation 20-sim can import an existing FMI 2.0 co-simulation FMU as a 20-sim block. 20-sim then acts as an FMI master simulator.

Export FMU export is supported in the following ways:

Tool Wrapper The 20-sim simulator has a built-in co-simulation feature based on XML-RPC calls. A tool wrapper FMU is developed for INTO-CPS that uses this interface to allow an FMI co-simulation with a 20-sim model running inside the 20-sim simulator. The FMU itself acts as an FMI wrapper for the existing XML-RPC co-simulation interface.

Standalone FMU export for FMI 1.0 and FMI 2.0 is supported in 20-sim since version 4.6. The FMU export template is based on

the platform-independent, standalone, ANSI-C source code template included in 20-sim. In contrast to the tool wrapper FMU, this FMU type has no dependencies on the 20-sim tool. FMI ModelExchange is not yet supported.

3D Animation 20-sim can export an existing 20-sim 3D animation as visualization FMU. This special type of FMU does not contain any model, but instead it shows a 3D animation window. This FMU only has inputs, and thus no outputs. Furthermore, this type of FMU is currently only supported on the Windows platform (32-bit and 64-bit). Next to this standalone 3D animation FMU, there is an ongoing development to create a cross-platform FMI visualization solution based on Unity [Tec16].

2.2.3 Mapping between 20-sim and FMI

Starting with 20-sim 4.6 it is possible to generate an FMU using a dedicated C-code generation template. The mapping from a submodel in 20-sim to an FMI description is a one-to-one translation of the sorted model equations to the corresponding ANSI-C code lines. The FMU code generation template adds an FMI 1.0 or 2.0 co-simulation interface around the generated code. Detailed information on the 20-sim code generation process can be found in deliverable D5.1d [HLG+15].

All 20-sim model inputs and outputs are mapped to FMI input and output variables. 20-sim parameters are mapped to FMI parameters and all 20-sim variables are mapped to FMI scalar variables. Matrices and vectors are flattened to a list of FMI scalar variables following the FMI structured naming convention [Blo14]. All 20-sim variables are accessible from the FMI interface. 20-sim C-code generation only supports variables of type double. This means that the available 20-sim types integer and boolean will also be converted to type double in the generated code. In the FMU interface, the original type as specified in the 20-sim model is used, which means that the FMI variable set and get functions will do a conversion to and from double values internally.

The 20-sim C-code generation feature was originally meant for real-time control applications. Therefore, all C-code generation templates, including the original year 1 FMU template, do not support variable step-size integration methods. File access and DLL functions are also not supported.

Since the co-simulation engine does not have (hard) real-time constraints,

the FMU code-generation template has been extended with variable step-size integration methods. Also, file access and DLL function calls have been added for use in a co-simulation. These features are specifically designed for the INTO-CPS project, and are at the time of writing only available in the INTO-CPS build of 20-sim 4.6.3.

CVODE and MeBDFi Integration Methods In the scope of the INTO-CPS project, 20-sim FMU generation is extended with the variable step-size integration methods CVODE Adams [HBG+05] and MeBDFi [Cas83, CC92]. This enables 20-sim to support a wider variety of models. Especially, systems with stiff equations [Wik16] and/or models containing algebraic relations can now be simulated in a more efficient way.

2D-table Reading Due to no real-time constraints in an FMU, file access is allowed in a co-simulation experiment. Based on INTO-CPS case study demands for reading external data files, reading a 2D-table from file has been implemented for FMU usage.

Unimplemented FMI Functionality Currently, only co-simulation FMI is supported. Model exchange FMI will not be supported in 20-sim within the scope of the INTO-CPS project.

An overview of the FMI functions that are not implemented is listed in Table 2. Because of limited string support, the FMI string mutator and accessor are not implemented. Furthermore, interpolation of input variables is not implemented. Therefore, the derivative functions for inputs and outputs are not implemented. The concept of FMU storage is only useful for model exchange and is not implemented in the current 20-sim FMI export for that reason. The last function in the FMI 2.0 list, "fmi2GetMaxStepSize", is a function that is proposed as a new function in the FMI standard [PBLG15]. It allows an FMU to report in advance the maximum step-size that it will accept in the next time step. The variable step-size calculator queries all FMUs for these step-sizes and uses the minimum of the reported values as upper bound for the next step-size.

The *UnitDefinitions*, *LogCategories* and *TypeDefinitions* are currently not implemented, but will be implemented in year three of the INTO-CPS project.

Table 2: FMI functions currently not implemented

FMI 1.0	**FMI 2.0**
fmiGetString	fmi2GetString
fmiGetStringStatus	fmi2GetStringStatus
fmiSetString	fmi2SetString
fmiGetRealInputDerivatives	fmi2GetRealInputDerivatives
fmiSetRealInputDerivatives	fmi2SetRealInputDerivatives
fmiGetRealOutputDerivatives	fmi2GetRealOutputDerivatives
	fmi2GetFMUstate
	fmi2SetFMUstate
	fmi2FreeFMUstate
	fmi2SerializedFMUstateSize
	fmi2SerializeFMUstate
	fmi2DeSerializeFMUstate
	fmi2GetDirectionalDerivative
	fmi2GetMaxStepSize

2.2.4 Additional Simulator Capabilities

Tool Wrapper Approach Instead of implementing the model internally in the FMU, the FMU can also interact with a running instance of the tool. This is called the tool wrapper approach. The tool wrapper approach has the advantage that the (sub)model that is co-simulated, can be inspected from within the tool itself. Also, results of a co-simulation can be inspected from the tool with which the FMU was generated. After the co-simulation has finished, the FMU will close the connection with the tool, but the simulation results can still be inspected within this tool.

External Monitoring To inspect the progress of a 20-sim simulation by an external application, 20-sim has been extended with functionality to transfer large sets of the latest simulation data on request. This so-called monitoring extension has been written specifically for the INTO-CPS project. This means that an external tool can register a list of monitor variables in 20-sim. During simulation, the external application can retrieve the values for these registered variables. The monitoring action itself does not interrupt the simulation in 20-sim. One application of this monitoring functionality is to pass 20-sim data to software that can visualise this data in a 3D scenery. A connection to Unity ([Tec16]), a game engine, is setup to animate a 3D-scenery based on variables obtained from 20-sim. Unity can make visually

appealing sceneries, and the visualisation can be done on multiple operating systems. This enables distributed visualisation on separate computers. This can be used for training simulator purposes, for example. The interface is built upon the XML-RPC interface, which enables other tools to use the monitoring functionality for their own purposes.

2.3 OpenModelica

OpenModelica [Fri04] is an open-source Modelica-based modeling and simulation environment. Modelica [FE98] is an object-oriented, equation-based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. The Modelica language (and OpenModelica) supports continuous, discrete and hybrid time simulations. OpenModelica always compiles Modelica models into FMU, C or C++ code for simulation.

Several integration solvers, of both fixed step and variable step-size, are available in OpenModelica: euler, rungekutta, dassl (default), radau5, radau3 and radau1. Currently, the FMU for co-simulation only supports the euler solver. Work is in progress to support more advanced solvers.

OpenModelica can be interfaced to other tools in serveral ways as described in the OpenModelica user's manual [Ope]:

- via command line invocation of the OpenModelica compiler (omc);
- via C API calls to the omc compiler dynamic library;
- via the CORBA interface;
- via OMPython interface [GFR+12].

OpenModelica has its own scripting language, Modelica script (mos files), which can be used to perform actions via the compiler API such as loading, compilation and simulation of models or plotting of results.

OpenModelica supports Windows, Linux and Mac Os X.

OpenModelica has support for static and dynamic debugging of Modelica models [PSA+14]. Static debugging helps the user understand how his model has been optimized and solved by the compiler via an equation browser. Dynamic debugging is currently available for algorithmic Modelica code and supports breakpoint-based debugging.

Debugging support in the generated FMUs is planned.

2.3.1 FMI Support

OpenModelica supports FMI 1.0 and FMI 2.0 export and import both for model-exchange and co-simulation. The FMI 2.0 export for co-simulation

Table 3: FMI functions currently not implemented

FMI 1.0	**FMI 2.0**
fmiGetRealInputDerivatives	fmi2GetRealInputDerivatives
fmiSetRealInputDerivatives	fmi2SetRealInputDerivatives
fmiGetRealOutputDerivatives	fmi2GetRealOutputDerivatives
	fmi2GetFMUstate
	fmi2SetFMUstate
	fmi2FreeFMUstate
	fmi2SerializedFMUstateSize
	fmi2SerializeFMUstate
	fmi2DeSerializeFMUstate
	fmi2GetDirectionalDerivative

has been implemented in Year 1 of the INTO-CPS project. In Year 2 the FMI 2.0 export has been has been improved and extended with support for Modelica resources (such as tables) and FMI 2.0 import in OpenModelica has been implemented. The new functionality is beeing tested and will be released with version 1.11 of OpenModelica during December 2016. The new functionality for exporting an FMU with Modelica resources (such as tables) packed inside it is needed for the Line Following Robot case study.

All OpenModelica generated FMUs are standalone. The FMI export reuses the OpenModelica templates available for C and C++ code generation.

An overview of the FMI functions that are not implemented is listed in Table 3. All the other functions that are given in the FMI standard are implemented.

3 Co-simulation Stability

Co-simulations are orchestrated using the COE, which is explained in detail in [LLW+15] including the its design. It basically uses the FMI functions available in the FMU's to perform steps in time as explained in [Blo14]. Most important is that all FMU's in a co-simulation advance time in parallel, then the port variables are communicated between the FMU's, and afterwards a new step can be taken. The step-size is determined by the COE, and should be such that the co-simulation is stable and assures a given accuracy.

It is important that the COE can measure the stability of the simulation. It can be used as an indication of how well the simulation is progressing, and how much the results can be trusted to say something about the real system under control. In the next sections, the work on measuring stability is described and how this relates to the concept of accuracy. As will be shown, the stability of the simulation can directly influence the step-size calculation of the co-simulation.

3.1 Stability vs. Accuracy

It is important to make a distinction between stability and accuracy. A co-simulation can be stable, but results may not be correct, due to an accuracy that is too low. Lowering the co-simulation step-size might improve the overall simulation in this case.

Another situation is that model in the co-simulation itself becomes unstable. This can be the result of a bad control algorithm. In this case, lowering the step-size of the co-simulation will not help stabilizing the simulation.

In [LLW+15] it is explained how the current step-size prediction of the COE works. It is based on four constraint types: Zero Crossing, Bounded Difference, Sampling rate and FMU-requested. The latter may be done by a newly proposed function `fmi2GetMaxStepSize` that may be implemented by an FMU.

The COE can adapt the step-size based on stability, since it can measure if the data of the co-simulation lies within predefined minimum and maximum boundaries. It however cannot account for accuracy in the computation of its step-size, since the COE has no means yet to evaluate if the results of the co-simulation are accurate enough. This accuracy aspect will be further investigated in year three of the INTO-CPS project.

3.2 Monitoring Stability

This section describes how stability can be measured and monitored using information from the FMUs. In Section 3.2.1, an approach is described where co-simulation invariants are placed in the output variables using information from the FMUs. Section 3.2.2 describes how this can be monitored.

3.2.1 Measuring using FMI Attributes

The current FMI 2.0 standard [Blo14] already has information that can be used to check stability of the co-simulation.
All variables in an FMU, including output variables, have a type. The integer type may define the optional attributes minimum and maximum, the real type extends the integer type with an optional nominal attribute. If these attributes are set, they can be used to check if the output variables are in range. If the variable goes out of range, it can be assumed that the co-simulation became unstable. The properties are defined in the attributes group for a ScalarVariable, `fmi2RealAttributes` and `fmi2IntegerAttributes`, for the `Real` and `Integer` types respectively.

Attributes from the integer type which can be used for stability checks:

min and max: Both Real and Integer types have the field `min` and `max` defined, where $max \geq min$.

Attributes from the real type which can be used for stability checks:

nominal: The Real type also has the attribute **nominal**. FMI 2.0 documentation says $nominal > 0$ is required. This has to do with the way the absolute tolerance for a variable is determined:

$$absoluteTolerance = nominal \cdot tolerance \cdot 0.01$$

So an additional requirement should be added for the actual value:

$$(min \leq nominal \leq max) \vee (min \leq -nominal \leq max)$$

unbounded: The Real type also has the attribute `unbounded`, which is a boolean indicating if the absolute value of a variable can get much larger than its nominal value. Typical examples are the monotonically increasing rotation angles of crank shafts and the longitudinal position of a vehicle along the track in long distance simulations. Variables that are unbounded should not be considered in the stability analysis.

derivative: A variable can also be marked as being the `derivative` of another `parent` variable. Also this `derivative` variable can have bounds, which says something about how fast the `parent` variable can change. The change of the `parent` variable can also be compared against the actual value of the `derivative`. If too far apart, instability is likely.

3.2.2 Measuring Internal FMU Stability

The options presented in Section 3.2.1 can be continuously checked during a simulation. For example, a value that lies outside of its minimum and maximum bounds can be detected.

An FMU may also become unstable because of its own internal dynamics. One way to determine if an FMU internally has difficulty in fulfilling its internal accuracy, is to measure the number of internal FMU model evaluations. However, currently the FMI 2.0 standard has no functions for this. A way to circumvent this lack in functionality is to measure the time it takes to compute an FMU step. If the time increases with respect to previous FMU steps, probably more internal steps are taken. This is, however, not a very reliable way of measuring, and therefore this has not been implemented yet.

3.3 Accuracy

The COE is able to measure the accuracy of the simulation. This can be used as one of the tools to adjust the step-size of the simulation such that a predefined accuracy is met. At first glance, this looks simple enough: take a step, determine the accuracy of the simulation, and if the accuracy is not within a predefined accuracy, redo the step. The problem with this approach is that not all FMU's might support backstepping. Whether an FMU supports backstepping can be determined by the attribute `canGetAndSetFMUstate` in the `fmiModelDescription` element in the `modelDescription.xml` file.

3.3.1 Measuring Accuracy Constraints

Currently an algorithm is implemented in the COE that uses zero-crossing constraints and the derivatives to adjust the step-size. It calculates deriva-

tives for up to order 2 to forecast the values. If the difference between the computed and forecasted values becomes too large with respect to the relative and absolute tolerance, the step-size is decreased. See the variable step-size algorithm described in [LLW+15] for more detail.

3.3.2 Measuring Accuracy using Multiple Simulations

If a co-simulation is run with different step-sizes, the output can be compared against eachother. The difference also says something about the accuracy of the simulation. If the results are close to each other, further decreasing the step-size will not have much effect.

3.3.3 Adjusting for Accuracy Problems Post Discovery

Rollback If the co-simulation engine determines instability, it is desired to rollback the simulation to a state where the simulation was still stable, and continue from that stored point with a smaller step-size. When rollback is not supported for a specific FMU, this FMU can be replayed from the beginning of the simulation with recorded inputs. At startup this is a good method to determine the proper step-size. When a simulation advances more in time, this replaying will take a lot of time though. This problem will be further investigated in year three of the INTO-CPS project.

Redo a Step by Changing the step-size When the COE is changing the step-size in case of a rollback, it needs to re-calculate all the values for all the FMU's in the co-simulation. It will not work if the COE only calls the FMU that triggered the rollback twice with a step-size of $\frac{1}{2}\ \Delta T$. This inputs of the faulty FMU will not be recomputed after the first half step. Therefore, the output after the second half step will be similar to the original computation with the full step.

4 Conclusions

In the second year of the INTO-CPS project the Co-simulation Orchestration Engine (COE) has been developed further. It can do a co-simulation with FMU's generated by the simulators Overture, 20-sim, and OpenModelica, as well as FMU's from other tools that support FMU export. Also a step-size adaptation scheme is implemented to support variable step-size co-simulation steps. An important aspect for varying the step-size of a co-simulation is to measure/monitor the stability of the co-simulation, which has been implemented in the COE.

With respect to 20-sim, UnitDefinitions and TypeDefinitions will be implemented in year three of the INTO-CPS project. Furthermore, more investigation will be done in the computation of output derivatives, and the use of input derivatives. These are used to achieve a more stable result. Another feature that will be implemented in 20-sim, is a tool wrapper for 3D animations, using Unity.

The Overture FMI integration will be extended with support for mapping fixed length VDM sequences as scalar variables. This is done by using the variable naming conversion *structured* from the FMI standard.

For stability during a co-simulation, more investigation will be done in year three at Linköping University.

References

[And16] Christian Andersson. Methods and tools for co-simulation of dynamic systems with the functional mock-up interface, 2016.

[Blo14] Torsten Blochwitz. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads`, July 2014.

[Cas83] J.R. Cash. The integration of stiff initial value problems in ODEs using modified extended backward differentiation formulae. *Computers & Mathematics with Applications*, 9(5):645–657, 1983.

[CC92] J.R. Cash and S. Considine. An MEBDF code for stiff initial value problems. *ACM Transactions on Mathematical Software*, 18(2):142–158, 1992.

[DES09] DESTECS (Design Support and Tooling for Embedded Control Software). European Research Project, June 2009. `http://www.destecs.org`.

[FE98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, January 2004.

[GFR+12] Anand Ganeson, Peter Fritzson, Olena Rogovchenko, Adeel Asghar, Martin Sjölund, and Andreas Pfeiffer. An OpenModelica Python interface and its use in pysimulator. In Martin Otter and Dirk Zimmer, editors, *Proceedings of the 9th International Modelica Conference*. Linköping University Electronic Press, September 2012.

[GMF12] C.J. Gamble, M. Mansfield, and J.S. Fitzgerald. The Co-Simulation of a Cardiac Pacemaker using VDM and 20-sim. In J. S. Fitzgerald, T. Mak, A. Romanovsky, and A. Yakovlev, editors, *Procs. Workshop on Trustworthy Cyber-Physical Systems*, volume CS-TR-1347 of *Technical Report Series*. School of Computing Science, Newcastle University, UK, 2012.

[HBG+05] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward.

SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[HLG+15] Miran Hasanagić, Peter Gorm Larsen, Marcel Groothuis, Despina Davoudani, Adrian Pop, Kenneth Lausdahl, and Victor Bandur. Design Principles for Code Generators. Technical report, INTO-CPS Deliverable, D5.1d, December 2015.

[IPG+12] Claire Ingram, Ken Pierce, Carl Gamble, Sune Wolff, Martin Peter Christensen, and Peter Gorm Larsen. Examples compendium. Technical report, The DESTECS Project (INFSO-ICT-248134), October 2012.

[KGD16] C. Kleijn, M.A. Groothuis, and H.G. Differ. *20-sim 4.6 Reference Manual*. Controllab Products B.V., 2016.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LLB11] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In Shengchao Qin and Zongyan Qiu, editors, *Proceedings of the 13th international conference on Formal methods and software engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[LLW+15] Kenneth Lausdahl, Peter Gorm Larsen, Sune Wolf, Victor Bandur, Anders Terkelsen, Miran Hasanagić, Casper Thule Hansen, Ken Pierce, Oliver Kotte, Adrian Pop, Etienne Brosse, Jörg Brauer, and Oliver Möller. Design of the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D4.1d, December 2015.

[LPH+15] Peter Gorm Larsen, Ken Pierce, Francois Hantry, Joey W. Coleman, Sune Wolff, Kenneth Lausdahl, Marcel Groothuis, Adrian Pop, Miran Hasanagić, Jörg Brauer, Etienne Brosse, Carl Gamble, Simon Foster, and Jim Woodcock. Requirements Report year 1. Technical report, INTO-CPS Deliverable, D7.3, December 2015.

[LRV+11] Kenneth G. Lausdahl, Augusto Ribeiro, Peter Visser, Frank Groen, Yunyun Ni, Jan F. Broenink, Angelica Mader, Joey W.

Coleman, and Peter Gorm Larsen. D3.3b — Co-simulation Foundations. Technical report, The DESTECS Project (INFSO-ICT-248134), December 2011.

[Ope] Open Source Modelica Consortium. OpenModelica User's Guide.

[PBLG15] Adrian Pop, Victor Bandur, Kenneth Lausdahl, and Frank Groen. Integration of Simulators using FMI. Technical report, INTO-CPS Deliverable, D4.1b, December 2015.

[PGP+16] Richard Payne, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 2. Technical report, INTO-CPS Deliverable, D3.5, December 2016.

[PSA+14] Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. Integrated Debugging of Modelica Models. *Modeling, Identification and Control*, 35(2):93–107, 2014.

[SBE+14] S. Sicklinger, V. Belsky, B. Engelmann, H. Elmqvist, H. Olsson, R. Wüchner, and K.-U. Bletzinger. Interface jacobian-based co-simulation. *International Journal for Numerical Methods in Engineering*, 98(6):418–444, 2014.

[Sie10] Alexander Siemers. *Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis.* Doctoral thesis No 1337, Linköping University, Department of Computer and Information Science, 2010.

[SNF05] Alexander Siemers, Iakov Nakhimovski, and Dag Fritzson. Metamodelling of mechanical systems with transmission line joints in modelica. In Gerhard Schmitz, editor, *Proceedings of the 4th International Modelica Conference*, March 2005.

[SP16] Stian Skjong and Eilif Pedersen. The theory of bond graphs in distributed systems and simulations. In *Proceedings of the 2016 International Conference On Bond Graph Modeling and Simulation - ICBGM'2016; 2016; Montreal; Quebec; Canada*, pages 147–157, 2016.

[Tec16] Unity Technologies. Unity. `https://unity3d.com/`, December 2016.

[Vie14] Antoine Viel. Implementing stabilized co-simulation of strongly coupled systems using the functional mock-up interface 2.0. In *Proceedings of the 10th International Modelica Conference; March*

10-12; 2014; Lund; Sweden, number 96, pages 213–223. Linköping University Electronic Press; Linköpings universitet, 2014.

[Wik16] Wikipedia. Stiff equation. `https://en.wikipedia.org/wiki/Stiff_equation`, June 2016.

A List of Acronyms

20-sim	Software package for modelling and simulation of dynamic systems
ACA	Automatic Co-model Analysis
AST	Abstract Syntax Tree
AU	Aarhus University
CLE	ClearSy
CLP	Controllab Products B.V.
COE	Co-simulation Orchestration Engine
CPS	Cyber-Physical Systems
CT	Continuous-Time
DE	Discrete Event
DESTECS	Design Support and Tooling for Embedded Control Software
DSE	Design Space Exploration
FMI	Functional Mockup Interface
FMI-Co	Functional Mockup Interface – for Co-simulation
FMI-ME	Functional Mockup Interface – Model Exchange
FMU	Functional Mockup Unit
HiL	Hardware-in-the-Loop
HMI	Human Machine Interface
HW	Hardware
ICT	Information Communication Technology
IDE	Integrated Design Environment
M&S	Modelling and Simulation
MBD	Model Based Design
MiL	Model-in-the-Loop
OMG	Object Management Group
OS	Operating System
PROV-N	The Provenance Notation
RPC	Remote Procedure Call
SiL	Software-in-the Loop
ST	Softeam
SVN	Subversion
SysML	Systems Modelling Language
TA	Test Automation
TRL	Technology Readiness Level
TWT	TWT GmbH Science & Innovation
UML	Unified Modelling Language
UNEW	University of Newcastle upon Tyne
UTP	Unifying Theories of Programming

UTRC	United Technologies Research Center
UY	University of York
VDM	Vienna Development Method
VSI	Verified Systems International
WP	Work Package
XML	Extensible Markup Language