Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Integration of simulators in the INTO-CPS Platform

Deliverable Number: D4.1b

Version: 0.3

Date: 2015

Public Document

http://into-cps.au.dk

Contributors:

Adrian Pop, LIU
Victor Bandur, AU
Kenneth Lausdahl, AU
Frank Groen, CLP

Editors:

Adrian Pop, LIU

Reviewers:

Carl Gamble, UNEW
Nuno Amálio, UY
Hassan Ridouane, 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	22-05-2015	Adrian Pop	Initial document version
0.2	26-08-2015	Adrian Pop	Restructure the document
0.3	15-12-2015	Adrian Pop	Final version changes

Abstract

This deliverable contains the design specification for integration of simulators (OpenModelica, Overture and 20-sim) with the INTO-CPS the co-simulation orchestration engine (COE) at the end of the first year of the project. The integration of the simulation tools into COE will use Functional Mockup Interface (FMI) with INTO-CPS extensions.

Contents

1 Introduction

This deliverable contains the design for the integration of simulators OpenModelica, Overture and 20-sim with the co-simulation orchestration engine (COE) using existing FMI and possible extensions to the standard (such as getting the maximum step size supported by a FMU, which has already been proposed in the literature).

The integrated simulators in these project are:

- OpenModelica [Fri04], `https://openmodelica.org`
- Overture [LBF+10], `http://overturetool.org/`
- 20-sim [Bro97], `http://www.20sim.com/`

1.1 The FMI standard

The Functional Mock-up Interface (FMI) [Blo14] is an open standard for exporting dynamic models on the system and component level either for model exchange or co-simulation. FMI gives model-based systems engineering the ability to share models between tools. More than 60 commercial or open-source simulation tools are now supporting the FMI standard.

Please see *D4.1d – Design of the INTO-CPS Platform* [LLW+15] for a description of the FMI standard.

1.2 Requirements

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

- 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

1.3 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 [SNF], [Sie10]. At the *model export level* for example Dymola can export Matlab S functions. Integration of Overture and 20-sim has been achieved [GMF12] before in the DESTECS [DES09], [LRV+11] EU project at the *simulator tool level*. Please see *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 FMI standard. The exported FMUs can then be co-simulated using the COE.

2 Integration of simulators

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

In this section we give the details of each simulator and its support for the FMI for co-simulation integration.

2.1 Overture

This section describes the VDM platform Overture from the point of view of its role in co-simulation scenarios.

2.1.1 Introduction

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.

Compliance with the standard makes Overture usable as one of the simulation engines in a complete FMI-based co-simulation scenario. This is achieved in two different ways:

- Overture can wrap a VDM-RT model in an FMI-compliant interface, and itself act as the simulation engine for the model. The result is a tool-wrapper FMU since the Overture tool is included together with the model for this kind of FMU. The FMI doStep function will then instruct the VDM interpreter to execute for a specific amount of time. The interpreter will then execute the number of expressions and statements that can complete within this time limit.
- Overture can translate VDM-RT models written in the executable subset of the language directly to compilable C code[2], wrapped in the appropriate FMI-compliant interface, to be used independently of Overture in a co-simulation scenario. The result is a standalone FMU because it does not rely on Overture and only consists of the model in an executable form, and the FMI functions required to interact with the model.

Both these features are discussed in further detail below in this document, and in deliverable[HLG+15].

2.1.2 FMI support

Overture FMU support is implemented as a tool wrapper, this means that exported models from Overture require the Overture tool to simulate. This is

[1] `www.eclipse.org`.
[2] This feature is currently under development.

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

implemented such that the VDM Interpreter and its FMI interface is included in the exported FMU. [3]

FMU export from Overture is currently done manually and supports the FMI 2.0 standard, with the exception of support for 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`. The export procedure currently consists of two tasks:

- Manual population of the `ModelDescription.xml` with scalar variables which are linked to either VDM-RT values or to VDM instance variables. The mapping rules are described in Section 2.1.3.
- Copying the `ModelDescription.xml` and VDM source files into the template FMU.

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.

[3]The Overture tool wrapper FMUs currently support Win32, Win64, Linux64, Darwin64 and require Java 1.7 to be installed and available in the PATH environment variable.

In its current state, the FMI export feature supports getting and setting Reals, Integers and Booleans, 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.

2.1.3 Mapping between VDM and FMI

The VDM language dialect which supports FMU export is object-oriented and thus does not have blocks with defined inputs and outputs. Therefore, a custom mapping of the internal system state to the scalar variables of FMI must be created manually. 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 Listing 2, valve actuator Listing 3, the controller Listing 4 and a system configuration Listing 1.

```
system System

instance variables

  public static controller : [Controller] := nil;
  levelSensor : LevelSensor;
  valveActuator : ValveActuator;
 ...
end System
```

Listing 1: The Watertank system

```
class LevelSensor

instance variables
  level : real := 0.0;
...
end LevelSensor
```

Listing 2: Level Sensor

```
class ValveActuator

instance variables
  valveState : real := 0.0;
..
end ValveActuator
```

Listing 3: Valve Actuator

```
class Controller

values

  public minLevel : real = 0.0; -- {m}
  public maxLevel : real = 0.0; -- {m}
...

end Controller
```

Listing 4: The Watertank controller

The mapping between VDM values (constants), instance variables, and FMI scalar variables, is shown in Table 2. The mapping includes the VDM value `maxLevel` and the instance variables `level` and `valveState`. This illustration only shows how basic, unstructured VDM values and instance variables can be mapped to FMI. Not all types of instance variable can be exported, since it must be possible to express a path to the variable from the system class using dots to separate fields, and the final element in the path may not be more than just the name of a variable. This means that instance variables and values may not come in the from of arrays, lists, sets *etc.*, as accessing leaf values stored in such variables would require the use of the corresponding accessor mechanism (array index for arrays, head element retrieval for lists *etc.*)[4]. The name mapping for Table 2 is given in Listing 5

The scalar variables must be mapped to the internal state of the VDM model as described above. This can be done using the model description element `Overture` illustrated in Table 2, where the scalar variable `valueReference` is linked to the corresponding path inside the System class of the VDM model.

```
<VendorAnnotations>
  <Tool name="Overture">
    <Overture>
      <link valueReference="0" name="Controller.maxLevel"/>
      <link valueReference="3" name="System.levelSensor.level"/>
      <link valueReference="4"
            name="System.valveActuator.valveState"/>
    </Overture>
  </Tool>
```

[4]The restriction on accessor mechanisms is due to a limitation of the current implementation and not the language.

Table 2: VDM to FMI mapping

VDM	FMI 2
class Controller -- Parameter **values** maxLevel: **real** = 5; **end** Controller	<ScalarVariable name="maxLevel" valueReference="0" causality="parameter" variability="fixed" initial="exact"> <Real start="5"/> </ScalarVariable>
class LevelSensor -- input **instance variables** level : **real** := 0.0; **end** LevelSensor	<ScalarVariable name="level" valueReference="3" causality="input" variability="continuous"> <Real start="0"/> </ScalarVariable>
class ValveActuator -- output **instance variables** valveState : **real** := 0.0; **end** ValveActuator	<ScalarVariable name="valveState" valueReference="4" causality="output" variability="discrete" initial="calculated"> <Real/> </ScalarVariable>

```
<VendorAnnotations>
```

Listing 5: Name mapping for the variables in Table 2

2.1.4 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 and input or write to an output. If the time it took to perform the calculation is less that 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 precisely matches the time when the write must occur. If not then `Discard` is returned from `doStep`.

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.

2.1.5 Additional simulator capabilities

The current version of the FMU export feature embeds the Overture interpreter inside the FMU and thus all tracing and logging features available to Overture can be used in the FMU. Future versions will allow the exported FMU to connect to a debug session in the Overture IDE, making the standard Overture debugger available.

2.2 20-sim

This section describes the simulator tool 20-sim and its FMI integration support.

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 integration method support, simulation back-stepping is available. External interfacing to 20-sim is available using scripting, custom DLL functionality and CSV file variables.

The C-code generation is designed for real-time control applications. To guarantee non-deterministic model execution time, variable step integration methods are not supported in the generated code. Furthermore, file access and optional DLL function calls are not implemented in the generated code. File access is not supported because this would violate (hard) real-time constraints. 20-sim generated C-code uses doubles for the model equation calculations. The variable types integer and boolean are not supported as data type, but are transfered to doubles. There is no rigid support for strings in the C-code generation process. Detailed information on the 20-sim code generation process can be found in [HLG+15].

2.2.2 FMI support

20-sim FMI support is implemented as a 20-sim C-code template and is thus reusing the 20-sim C-code generation framework. The execution model of the 20-sim model is stored in 32-bits and 64-bits windows DLL's, and the meta model is represented in the FMI modelDescription XML file. Furthermore the C-code is exported to the generated FMU unit.

FMU generation can be done for the FMI 1.0 and FMI 2.0 standard. Currently only Cosimulation FMI is supported. The current state of the FMI export does support integer, string and boolean types on the FMI interface, while these type of variables are converted to double in the underlying model.

An overview of the FMI functions that are not implemented is listed in Table 3. Because of limited string support, the FMI string mutator and ac-

Table 3: 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

cessor are not implemented. Furthermore interpolation of input variables is not implemented and therefore the derivative functions for inputs and outputs are not implemented. The concept of FMU storage is only useful for ModelExchange and is therefore not implemented in the current 20-sim FMI export.

The FMI model description XML file provides in a mapping of all 20-sim model variables to their FMI counter part. All variables are accessible from the FMI interface. The *UnitDefinitions* and *TypeDefinitions* are currently not implemented but will be implemented later. FMI 2.0 also supports *LogCategories* to present FMU status information to the co-simulator in a nice manner but this is currently not implemented.

2.2.3 Additional simulator capabilities

The 20-sim simulator provides in plot functionality where variables can be plotted using multiple plot windows. To visualize (3D) simulations 3D animation is available. Furthermore the simulator provides in break point functionality which allows for convenient model debugging. For the scope of system integration it might be beneficial to reuse these functionalities in a FMI co-simulation fashion.

2.3 OpenModelica

This section describes the simulator tool OpenModelica and its FMI integration support.

2.3.1 Introduction

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 both fixed step and variable step size are available in OpenModelica: euler, rungekutta, dassl (default), radau5, radau3 and radau1.

OpenModelica can be interfaced to other tools in serveral ways as described in the OpenModelcia 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.

2.3.2 FMI support

OpenModelica supports FMI 1.0 and FMI 2.0 export and import both for model-exchange and co-simulation. All OpenModelica generated FMUs are standalone. The FMI export reuses the OpenModelica templates available for C and C++ code generation.

Table 4: FMI functions currently not implemented

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

An overview of the FMI functions that are not implemented is listed in Table 4.

2.3.3 Additional simulator capabilities

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 for the near future.

2.4 Into-CPS FMI extensions

In this section we shall list the FMI standard extension functions. Currently only `getMaxStepsize` is proposed. The function returns the maximum step size that the FMU can take. This information can be used by the COE to speed up the co-simulation.

```
fmi2Status getMaxStepsize(fmi2Component comp, fmi2Real* time);
```

3 Conclusions

In the first year of the INTO-CPS project the focus was on the integration of simulators (baseline tools) with the COE which has now be achieved as described in this document.

Each of the simulators Overture, 20-sim, OpenModelica can now export FMUs that can be simulated via the COE.

References

[Blo14] Torsten Blochwitz. Functional mock-up interface for model exchange and co-simulation. https://www.fmi-standard.org/downloads, July 2014. Torsten Blochwitz Editor.

[Bro97] Jan F. Broenink. Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25, 1997.

[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.

[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.

[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, Anders Terkelsen, Miran Hasanagić, Casper Thule Hansen, Carl Gamble, Oliver Kotte, Adrian Pop, and Etienne Brosse. 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, and Simon Foster. 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.

[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.

[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.

[SNF] Alexander Siemers, Iakov Nakhimovski, and Dag Fritzson. Metamodelling of mechanical systems with transmission line joints in modelica.

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