Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs

Casper Thule[1], Kenneth Lausdahl[2], and Peter Gorm Larsen[1]

[1] Aarhus University, Department of Engineering
Finlandsgade 22, 8200 Aarhus N, Denmark
{casper.thule,pgl}@eng.au.dk
[2] Mjølner Informatics A/S
Finlandsgade 10, 8200 Aarhus N, Denmark
kgl@mjolner.dk

Abstract. The Functional Mock-up Interface is a standard for co-simulation, which both defines and describes a set of C interfaces that a simulation unit, a Functional Mock-up Unit (FMU), must adhere to in order to participate in such a co-simulation. To avoid the effort of implementing the low level details of the C interface when developing an FMU, one can use the Overture tool and the language VDM-RT. VDM-RT is a VDM dialect used for modelling real-time and potentially distributed systems. By using the Overture extension, called Overture FMU, the VDM-RT dialect can be used to develop FMUs. This raises the abstraction level of the implementation language and avoids implementation details of the FMI-interface thereby supporting rapid prototyping of FMUs. Furthermore, it enables precise time detection of changes in outputs, as every expression and statement in VDM-RT is associated with a "timing cost". The Overture FMU has been used in several industrial case studies, and this paper describes how the Overture tool-wrapper FMU engages in a co-simulation in terms of architecture, synchronisation and execution. Furthermore, a small example is presented.

Keywords: Overture, Functional Mock-up Interface, VDM-RT, Co-Simulation, Real-time, Discrete-Event

1 Introduction

In general, co-simulation enables different constituent models, which form a coupled system, to be modelled in a distributed manner and then simulated in collaboration [5,6]. Hence, the modelling is carried out at the constituent model level without a detailed understanding of the other constituent models. A challenge in co-simulation is to synchronise the different simulating units ensuring that the timing of the overall simulation is sufficiently close to how this would work in reality. In such co-simulations it is often convenient to combine Discrete Event (DE) formalisms (typically describing cyber control aspects) with Continuous-Time (CT) formalisms (usually describing physical phenomena being controlled). Enabling such hybrid combinations generally require some kind of coordination and in this paper the focus is on the Functional Mock-up Interface (FMI) standard.

The contribution of this paper is to enable VDM-RT models to be exported as FMUs such that these models can be incorporated in a setting where some of the constituent models are made using Overture while others are made other other tools supporting FMI version 2.0. In this way the extension described here extend the places where Overture can be used in a CPS context for DE models. This provides a more abstract language for modelling and developing FMUs as opposed to implementing them in a native language, which can be beneficial in the systems engineering process [8].

In Sect. 2 we provide a short introduction to the background of this work. Next, Sect. 3 demonstrates how Overture can be used to produce FMUs by means of a small case study. Section 4 presents an overview of the architecture of this capability. Finally, Sect. 5 gives a few concluding remarks.

2 Background

The VDM-RT dialect historically started off as a notation called "VDM In Constrained Environments" (VICE) [18]. However, VICE performed poorly in the analysis of distributed systems [22]. Thus the notation was rethought, and extended with support for distribution and called VDM-RT [24]. Initial work with co-simulation using VDM-RT and 20-sim was carried out in Marcel Verhoef's PhD thesis [23]. VDM-RT was then further developed in a co-simulation context inside the DESTECS project [2]. The main result here was the Crescendo tool [14] combining DE formalism VDM-RT [10] with the CT formalism bond graphs using the 20-sim tool [9]. The co-simulation carried out here was bespoke and worked in general between these two tools. However, it was also demonstrated in DESTECS that it was possible to use Matlab/Simulink instead of 20-sim via an XML-RPC interface (revisited in Sect. 4).

Subsequently the INTO-CPS project [4] took this further using the FMI standard to achieve an open tool chain enabling any modelling and simulation tool able to produce Functional Mock-up Units using version 2.0 of the standard to be co-simulated [12]. The coordination of this co-simulation is performed by the INTO-CPS Co-simulation Orchestration Engine called Maestro [21]. FMI is a result of the MODELISAR project [7] and it is a tool independent standard for model exchange and co-simulation, where we only concern ourselves with the co-simulation part in this article. FMI defines a C interface that simulation units must implement in order to participate in a co-simulation. A simulation unit implementing the FMI interface is called a Functional Mock-up Unit (FMU). Such an FMU is packaged as a Zip archive, which contains libraries for the platforms that the executable part of the FMU has been compiled for, a model description file describing the scalar variables and their causality (input, output, parameter etc.) of the FMU, and a resources folder containing elements used internally by the libraries. The iteration carried out by a co-simulation master is roughly equivalent to getting inputs, setting outputs, and invoking the FMUs to progress for a determined step size. The process is repeated until a predetermined end time is reached.

An extension to FMI that adds an additional function to the interface called `GetMax-StepSize` has been proposed [3]. The purpose is that each FMU can be queried for the maximum step it can perform, and then the chosen step size is the minimum of all the reported step sizes, as an FMU always must be able to perform a smaller step than the reported maximum step. This makes it possible to avoid rolling back FMUs by setting a previous retrieved state, a feature not supported by VDM-RT FMUs. Furthermore, it makes it possible to synchronise at the specific point in time where an output is changed by a DE FMU. Besides detecting the point in time to synchronise it also makes it unnecessary to execute the co-simulation with a small time step to ensure detection of the changes in outputs, as the proper step size is reported by the FMU in question. The querying for maximum step sizes by the co-simulation master would occur after setting inputs and before invoking the FMUs to progress in the iteration described above.

3 Developing an FMU with Overture

In this section the FMI additions to a VDM-RT project required to export the project as an FMU is presented. This presentation concerns the structure of a VDM-RT project in order to be exported as an FMU and the template generated by the plugin. Afterwards, an example of a co-simulation is demonstrated where one of the FMUs is generated by using the plugin. Overture can generate both tool-wrapper FMUs and source code FMUs [1]. In this paper we focus on tool-wrapper FMUs.

3.1 FMI Additions for VDM-RT using Overture FMU

The Overture FMU plugin contains functionality to automatically generate a project template that complies with the required structure. This template and thereby the required structure is the following:

- A VDM-RT system `System` containing the definition of a given system by describing how different parts are deployed to different Core Processing Units (CPUs) [13]. This is not a class, but a system. The syntax is similar to ordinary classes with some differences, for example that it cannot be instantiated.
- A conventional [11] VDM-RT class `World` that provides an entry point into the model.
- A VDM-RT class called `HardwareInterface`, which is exemplified below in Sect. 3.3. This class contains the definition of the ports of the FMU. Its structure is enforced, and a self-documenting annotation scheme[1] is used such that the `HardwareInterface` class may be hand-written. The annotation format is `-- @ interface: type = [input/output/parameter/local], name="...";` and must be located directly above a **`value`** or an **`instance variable`** of one of the subclasses of the `Port` class described below. Ports of type parameter must be **`values`**, and all other ports must be **`instance variables`** of the class `HardwareInterface`. The reason for this approach is to capture all assumptions about FMI in the VDM-RT model itself opposed to extending the VDM-RT language or providing addition configuration files. This provides a solution where the generic FMI interface can be defined in a library and any instantiation

[1] The annotation scheme is documented on the INTO-CPS website http://into-cps-association.GitHub.io under "*Constituent Model Development → Overture → FMU Import/Export*.

hereof can be type checked with the concrete specification. Annotations must be provided since the FMI causality cannot be deduced automatically. Furthermore, these annotations convey the causality of the ports and has not resulted in any changes of the VDM-RT language, since they are written in comments.

– The library file `Fmi.vdmrt` defines the hardware interface port types used in `HardwareInterface`. This file contains an inheritance structure with a top-level generic `Port` class that is subclassed by ports for each FMI type: Bool, Real, Int and String. These subclasses are constructed with an initial value and contain get and set methods. Part of the class is presented in Listing 1; it also contains `StringPort`, `RealPort` and `BoolPort` implemented in a similar fashion to `IntPort`. The `getValue` function is declared **`pure`**, which means it does not update state (and other constraints described in [13].

```
class Port

types
  public String = seq of char;
  public FmiPortType = bool | real | int | String;

operations
  public setValue : FmiPortType ==> ()
  setValue(v) == is subclass responsibility;

  public pure getValue : () ==> FmiPortType
  getValue() == is subclass responsibility;

end Port

class IntPort is subclass of Port

instance variables
  value: int:=0;

operations
  public IntPort: int ==> IntPort
  IntPort(v)==setValue(v);

  public setValue : int ==> ()
  setValue(v) ==value :=v;

  public pure getValue : () ==> int
  getValue() == return value;

end IntPort
```

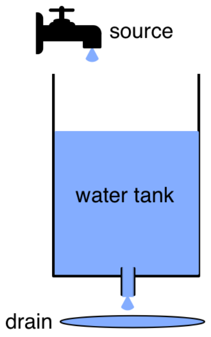
Listing 1: FMI library for VDM-RT containing `Port` class and subclasses for each FMI type.

After this required project structure is set up, the behaviour of the FMU must be implemented, which is demonstrated in the following section.

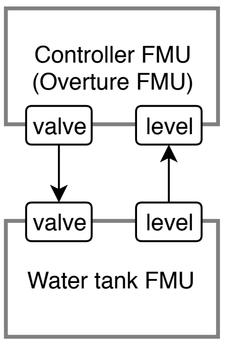
3.2 Overture FMU Example

In this section the controller of a water tank [17] shown in Fig. 1a is presented[2]. Afterwards, it is described in Sect. 3.4 how the FMI extension `GetMaxStepSize` maps to a VDM-RT model. Finally, Sect. 3.5 presents the results of a co-simulation where the VDM-RT model and Overture FMU is used.

The water tank is equipped with a source of water, two sensors, representing minimum and maximum water level, and a valve. When the valve is open, water pours out of the tank, and when the valve is closed, the water level rises, as the water is still flowing from the source. The water level is regulated by a controller expressed in a DE model using Overture and VDM-RT, which models the actuator that opens the valve when a maximum water level is reached and closes the valve when a minimum water level is reached. The draining and filling of the water tank and thereby the water level is expressed in a CT model described in [17]. The FMUs, their ports and dependencies between them are shown in Fig. 1b.



(a) Water tank example [17]



(b) Water tank FMUs

Fig. 1: Overview of the water tank

3.3 VDM-RT Model

The model realisation in VDM-RT is structured as presented in Fig. 2, which matches the description of the template in Sect. 3.1. The realisation is described below.

[2] The other FMU describing the CT part of the water tank is not described here; the interested reader is referred to [17].

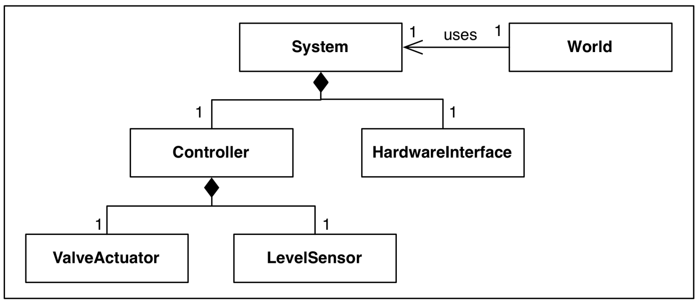


Fig. 2: Architecture of the VDM-RT model [17]

The `HardwareInterface` class is the interface of the DE model. In order to determine this interface it is necessary to consider the entire water tank system. As the state of the valve is operated by the DE model and has an impact on the calculation of the water level performed by the CT model, it must be an output from the DE model to an input on the CT model. The water level is calculated by the CT model and the DE model requires this information to determine whether to open or close the valve, and therefore it is an output from the CT model to an input on the DE model. Furthermore, it is necessary with two parameters on the DE model describing the minimum and maximum water level, as the DE model controls the state of the valve. The source of water is embedded in the CT model and therefore not considered. This leads to the interface presented in Listing 2, where the valve state has the type `BoolPort`, the water level has the type `RealPort`, and the parameters have the type `RealPort`. The value of the parameters can initially be changed by the co-simulation master based on the co-simulation configuration.

```
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);

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

end HardwareInterface
```

Listing 2: The hardware interface of the water tank controller

The LevelSensor class in Listing 3 encapsulates the port representing the level input. Notice that the set method is absent as level is an input, and therefore it is only possible to read a value from the port.

```
class LevelSensor
instance variables
  port : RealPort;
operations
  public LevelSensor: RealPort ==> LevelSensor
  LevelSensor(p) == port := p;

  public getLevel: () ==> real
  getLevel()== return port.getValue();
end LevelSensor
```

Listing 3: The encapsulation class for the water level sensor

The ValveActuator class in Listing 4 is similar in structure to the LevelSensor described above, but it captures the valve output instead of an input. It follows that the get method is absent, as valve is an output, and therefore it is only possible to write a value to the port.

```
class ValveActuator
instance variables
  port : BoolPort;
operations
  public ValveActuator: Port ==> ValveActuator
  ValveActuator(p) == port := p;

  public setValve: bool ==> ()
  setValve(value)== port.setValue(value);
end ValveActuator
```

Listing 4: The encapsulation class for the valve actuator

The Controller class in Listing 5 is the core logic of the DE model. It is instantiated with the LevelSensor and ValveActuator instances described above. The behaviour is contained in the loop operation, which takes 2 cycles[3] and runs every 10

[3] A cycles or duration statement at the top level of the loop operation as in this case can lead to undesired behaviour. Everything within the body of the cycles statement is executed atomically with the given cycle number, and thus prevents the scheduler from swapping out the current atomic block. As a result, periodic threads will not have the next period thread swapped before the current is completed. Therefore, no overlapping errors will be raised because the next period threads are not yet executing, even though the period has elapsed. This can be seen by setting the CPU frequency in Listing 6 to e.g. 20 Hz, thereby the cycles would take 50 milliseconds, but the period is still 10 milliseconds but no error is reported.

milliseconds until the simulation terminates. 2 cycles in this case corresponds to exactly 10 milliseconds and is calculated as:

$$\tau = cycles/freq_{CPU} = 2/200Hz = 0.01seconds$$

where τ is time, $cycles$ is the number of cycles from Listing 5, and $freq_{CPU}$ is the CPU frequency from Listing 6.

The behaviour of the loop operation is first to read the level, check whether it is above the maximum level or below the minimum level and open or close the valve respectively.

```
class Controller

instance variables
  levelSensor   : LevelSensor;
  valveActuator : ValveActuator;

values
  open : bool = true;
  close: bool = false;

operations
  public Controller : LevelSensor * ValveActuator  ==> Controller
  Controller(l,v)==
  (
    levelSensor   := l;
    valveActuator := v;
  );

  private loop : () ==>()
  loop()==
    cycles(2)
    ( let level : real = levelSensor.getLevel() in
      ( if( level >= HardwareInterface`maxlevel.getValue())
        then valveActuator.setValve(open);

        if( level <= HardwareInterface`minlevel.getValue())
      then valveActuator.setValve(close); );
    );

thread
  periodic(10E6,0,0,0)(loop);

end Controller
```

Listing 5: The `Controller` class with the core logic

The system entity `System` shown in Listing 6 is responsible for describing how the `controller` class of the water tank controller is deployed to a CPU and how it is connected to other parts in the model. Therefore `System` instantiates the hardware interface, instantiates and initialises the hardware encapsulation classes and passes these to the `Controller`, which is also instantiated and deployed on a CPU.

```
system System

instance variables
  -- Hardware interface variable required by FMU Import/Export
  public static hwi: HardwareInterface:= new HardwareInterface();
  public levelSensor : LevelSensor;
  public valveActuator : ValveActuator;
  public static controller : [Controller] := nil;
  cpu1 : CPU := new CPU(<FP>, 200);

operations
public System : () ==> System
  System () ==
  ( levelSensor := new LevelSensor(hwi.level);
    valveActuator :=  new ValveActuator(hwi.valveState);
    controller := new Controller(levelSensor, valveActuator);
    cpu1.deploy(controller,"Controller");
  );

end System
```

Listing 6: System class of the DE model

The `World` class launches the simulation by invoking the **`start`** statement in the `Controller` class instance, which is contained in `System` described above. This leads to the thread contained within the `Controller` class described above to be started. The implementation of the `World` class is presented below in Listing 7.

```
class World
operations
  public run : () ==> ()
  run() ==
  ( start(System`controller);
    block(); );

  private block : () ==>()
  block() == skip;

sync
  per block => false;
end World
```

Listing 7: World class of the DE model

3.4 Synchronisation

Synchronisation in terms of FMI is when outputs are exchanged. From an Overture FMU perspective, the synchronisation should ideally occur just before reading a value from a port and after writing to a port. This ensures synchronisation exactly when an output has changed or allows for retrieving updated inputs just before an input is read. Listing 8 shows an implementation of the `loop` operation, where the cycles statement is removed and thereby expressions and statements takes 2 cycles. This allows synchronisation at the desired synchronisation points, which `GetMaxStepSize` will return. The VDM-RT interpreter makes use of transactions [16], in the sense that it calculates the behaviour until the next synchronisation point, but does not commit it and thereby it is not exposed until the correct point in time. Thereby it is possible to calculate the value that `GetMaxStepSize` as the minimum time of all transactions as:

$$min(\{\tau | (\Sigma, \tau) \in T\}) - \tau_{now}$$

where (Σ, τ) is a transaction pair of state Σ to expose at time τ, T is a set of all transactions across CPUs, and τ_{now} is the global current time of the co-simulation.

```
private loop : () ==>()
loop()==
   -- SYNCHRONISATION
   let level : real = levelSensor.getLevel() in
   (  if( level >= HardwareInterface`maxlevel.getValue())
      then valveActuator.setValve(open);
      -- SYNCHRONISATION if condition yields true

      if( level <= HardwareInterface`minlevel.getValue())
      then valveActuator.setValve(close);
      -- SYNCHRONISATION if condition yields true
  );
```

Listing 8: Control loop the DE model with desired synchronisations.

3.5 Co-simulation Result

The result of a co-simulation of the water tank using the VDM-RT model is presented in Fig. 3. It shows that when the water level exceeds the maximum water level of two the valve opens, represented by the value 1. It remains open until the water level is below the minimum water level of one, at which point the valve is closed represented by the value 0. The step size of the co-simulation is 0.1 seconds. The small step delay is a result of the Jacobian master algorithm [19] used by the employed co-simulation orchestration engine. Instead one could use the Gauss-Seidel [19] master algorithm. This particular issue is addressed in [20].

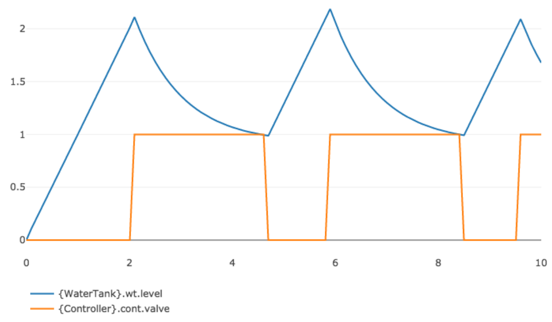


Fig. 3: Result of a co-simulation of the water tank

4 Architecture of Overture FMU

The architecture of the Overture FMU and the flow of messages is shown in Fig. 4[4]. The Overture FMU product is split into three parts that communicate via shared memory and protobuf messages. The first part, FMU, defines the FMU library that is invoked by the co-simulation master. The next part, Shared Memory (SHM), are the libraries involved in converting the data to Protobuf messages and using shared memory to pass data from the co-simulation Master to the last part, Model Execution, and back. The third and last part, Model Execution, describes the functionality that carries out the simulation of the model, where the Java application Overture-FMU[5] essentially is a Crescendo implementation [15] with a different protocol. The reason for this structuring is, that the SHM part is implemented in such a way, that it could easily be adapted to contain other messages, that are not FMI-specific. The three parts are described below along with the loading process of an Overture FMU and transferring of messages between the different parts.

In order to understand this section, some terminology must be known:

Protobuf: Protobuf[6] is the short name for Protocol Buffers, which is a language and platform-neutral extensible mechanism for serialising structured data developed by Google. It supports Java and C++ among others, which is used in the development of the Overture FMU described in Sect. 4.

Java Native Interface (JNI): JNI is a framework that makes it possible for Java applications to communicate with native libraries. This is also used in the Overture FMU.

[4] The libraries mentioned in the figure is part of the contribution of the work presented in this paper, unless explicitly stated otherwise.

[5] Notice the hyphen, which differentiates the application (Overture-FMU) from the name of the product (Overture FMU).

[6] Available at https://developers.google.com/protocol-buffers/

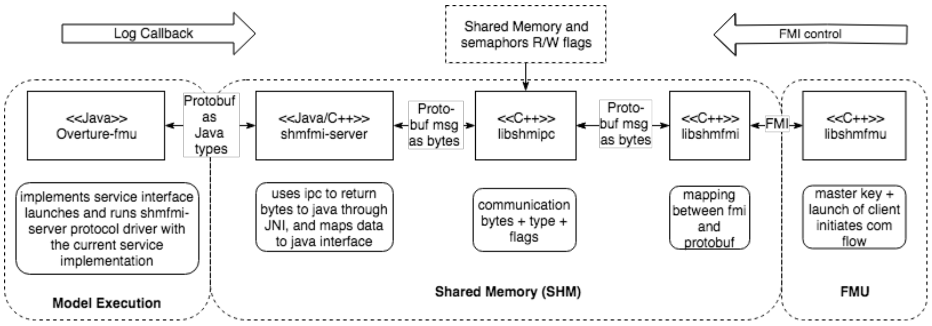

Log Callback
Shared Memory and semaphors R/W flags
FMI control
<<Java>> Overture-fmu
Protobuf as Java types
<<Java/C++>> shmfmi-server
Proto-buf msg as bytes
<<C++>> libshmipc
Proto-buf msg as bytes
<<C++>> libshmfmi
FMI
<<C++>> libshmfmu
implements service interface launches and runs shmfmi-server protocol driver with the current service implementation
uses ipc to return bytes to java through JNI, and maps data to java interface
communication bytes + type + flags
mapping between fmi and protobuf
master key + launch of client initiates com flow
Model Execution
Shared Memory (SHM)
FMU

Fig. 4: Architecture of Overture FMU split in three parts parts: FMU, Shared Memory (SHM), and Model Execution.

4.1 FMU

The overall responsibility of the FMU block[7] is to provide a C implementation of the FMI interface allowing it to serve as an FMU. libshmfmu is therefore the FMU implementation adhering to the FMI interface that is loaded when Maestro loads the library inside the FMU. It initialises the communication flow, instantiates the SHM libraries libshmfmi and libshmipc, and launches Overture-FMU, which is described in Sect. 4.3. When a function of FMU is invoked, the function invocation information is passed to the libshmfmi within SHM. The master key mentioned in the figure is a session key ensuring that multiple Overture tool-wrapper FMUs can coexist without interfering with each other.

4.2 Shared Memory (SHM)

The SHM part is responsible for passing messages between Model Execution and FMU using shared memory. This involves mapping each FMI function invocation to protobuf messages and the other way around. These messages are sent through a fixed-size shared memory space with read/write access being controlled by several semaphores. This native part had to be developed almost without any frameworks, as most of the frameworks suitable for the task could not be cross compiled with a reasonable effort. It was challenging to ensure cross compilation, which is an important feature. Concretely, the implementation is split into three libraries that are described below:

libshmfmi: This library does the mapping of FMI function invocations to protobuf messages, which is stored in the shared memory. Furthermore, it maps the response from protobuf messages to FMI.

libshmipc: This library embeds the shared memory required for communication and two semaphores to control access.

shmfmi-server: The shmfmi-server contains the bridge between Java and native code required to extract bytes from the shared memory, convert them to protobuf messages

[7] The source code is available at https://github.com/overturetool/shm-fmi.

and invoke the relevant functions in Overture-FMU described below. It invokes the relevant functions by exposing a Java interface, which defines the FMI calls with protobuf data types, that is implemented by CrescendoFMU, which is part of Overture-FMU. This Java Interface defines the FMI calls with protobuf data types. Furthermore, it also performs the mapping the other way with replies. Note, that this also operates in reverse, as there are callbacks in FMI e.g. for logging. Conceptually this is simple, but the implementation at the low level is not trivial. Concretely, shmfmi-server extracts bytes from the shared memory and embeds the JNI interface, which enables access from Java. It is instantiated by Overture-FMU in the Model Execution part.

As mentioned in Sect. 2 Crescendo features an XML-RPC interface, which is an alternative to this shared memory approach. The reason for choosing to use shared memory is that XML-RPC uses XML and a socket, which is slower in terms of performance than Java or native C calls. Furthermore, Crescendo did not feature `GetMaxStepSize`, so the Crescendo protocol would have to be changed in order to achieve the same functionality. Additionally, Crescendo was co-simulation between two instances and not N instances, so the slow down per simulator would be significant. Knowing that the shared memory approach is faster, it is a better solution in this case. It was also envisioned that the SHM functionality can be reused for other projects and for the C code generator described in [1], which it was not unfortunately.

4.3 Model Execution

This represents the left-hand side of Fig. 4 and thus the actual execution of the VDM-RT model. It contains the Java application Overture-FMU[8] that is launched by libshmfmu and acts as an interface to the Crescendo implementation. As mentioned in Sect. 2 VDM-RT was used in a co-simulation context in the DESTECS project, and therefore the main functionality of participating in a co-simulation was already present. It was therefore of interest to preserve most of the main functionality, but it was necessary to make changes to the interface in order to support FMI. The extension was realised by exposing the simulation driver of Crescendo, thereby enabling overriding. Thus Overture-FMU is an application that interprets FMI messages and utilises Crescendo to execute the model, and then adapts the Crescendo response to FMI. A low level detail is, that the VDM-RT interpreter [16] is packaged inside the resources folder of an FMU, mentioned in Sect. 2. The significant development additions in order to perform this adaption consist of the elements described below:

New Entry Point (main): This receives the master key to the shared memory as an argument and connects to the existing SHM via shmfmi-server described above. Afterwards, it instantiates the CrescendoFMU described below.

Mediation Between FMI and Crescendo (CrescendoFMU): This invokes the FMI-SimulationManager to perform a given task based on the protobuf message converted to Java by shmfmi-server. Afterwards, it creates an FMI Protobuf reply, which is returned to shmfmi-server. It implements the interface described in shmfmi-server above.

[8] The source code is available at https://github.com/overturetool/overture-fmu.

Extension of the SimulationManager (FMISimulationManager): In Crescendo the controlling entity was inherent in the VDM-RT resource scheduler, but this is not the case when using FMI, where Maestro is the controlling entity, and therefore additional changes had to be carried out. Furthermore, the synchronisation is different. The FMISimulationManager ensures that the interpreter only calculates to a certain time. It will calculate until just before it needs to read an input and right after an output, as this is where the synchronisation should occur. This way the interpreter is "ahead" of the global co-simulation time, as described in Sect. 3.4, but capable of calculating a value for `GetMaxStepSize` presented in Sect. 2.

Handling State (StateCache): The StateCache is necessary because of the way Crescendo operates, where the execution of a step takes all inputs and returns all outputs. However, FMI operates differently where the `setinput`, `dostep`, and `getoutput` calls are separated. Therefore this cache was added in order to support FMI.

5 Concluding Remarks

An advantage of a tool-wrapper FMU as opposed to source code / native FMUs is that the model is interpreted exactly as the language describes and modifications to the VDM-RT language are available out of the box. Furthermore, it allowed reuse of the existing tooling without changes to the interpreter. A disadvantage is, that it requires prerequisites to execute a tool-wrapper FMU, e.g. that Java is installed and available, and it is most likely slower in terms of performance compared to a native FMU. Future work on the Overture FMU tool-wrapper is to test with multiple FMI engines and publish the results on the FMI tools website [9]. Additionally, it would be interesting to perform benchmarks and comparisons with other FMUs. A new version of the FMI Standard is also under development, and Overture FMU should be updated to support this.

In this article it has been shown how one can use VDM-RT and Overture to develop a tool-wrapper FMU that can participate in an FMI co-simulation. This has been exemplified by realising a DE controller of a water tank system using Overture FMU and co-simulating it. Additionally, the architecture of a tool-wrapper Overture FMU has been described in depth. It contains native libraries, as an FMU must expose a C interface, that communicates with other shared native libraries over shared memory protected by semaphores. Furthermore, this also involves launching a Java application that reuses functionality from the Crescendo tool [2]. Emphasis has also been placed on describing how calculation of step sizes and synchronisation is carried out, as Overture FMU is unique in this field. Overture FMU has been successfully used in several industrial case studies as part of the INTO-CPS project [4].

Acknowledgments The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. We would like to thank all the participants of those projects for their efforts making this a reality. Furthermore, we would like to thank the anonymous reviewers for their comments, which helped to improve the paper.

[9] Available at http://fmi-standard.org/tools/

References

1. Bandur, V., Tran-Jørgensen, P.W.V., Hasanagic, M., Lausdahl, K.: Code-generating VDM for Embedded Devices. In: Fitzgerald, J., Tran-Jørgensen, P.W.V., Oda, T. (eds.) Proceedings of the 15th Overture Workshop. pp. 1–15. Newcastle University, Computing Science. Technical Report Series. CS-TR- 1513 (September 2017)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., Wouters, F.: Design Support and Tooling for Dependable Embedded Control Software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. pp. 77–82. ACM (April 2010)
3. Broman, D., Brooks, C., Greenberg, L., Lee, E., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of FMUs for co-simulation. In: Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on. pp. 1–12 (2013)
4. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)
5. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. Accepted on January 11, 2018 for publication in ACM Computing Surveys
6. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), http://arxiv.org/abs/1702.00686
7. ITEA Office Association: Itea 3 project, 07006 modelisar. https://itea3.org/project/modelisar.html (December 2015)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Heyward Street, Cambridge, MA02142, USA (April 2006), iSBN-10: 0-262-10114-9
9. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
10. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
11. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
12. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
13. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (April 2010)
14. Larsen, P.G., Lausdahl, K., Coleman, J., Wolff, S., Kleijn, C., Groen, F.: Crescendo Tool Support: User Manual. Tech. Rep. TR-001, The Crescendo Initiative, www.crescendotool.org (November 2013)
15. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Towards a co-simulation semantics of VDM-RT/Overture and 20-sim. In: Plat, N., Nielsen, C.B., Riddle, S. (eds.) Proceedings of the 10th Overture Workshop. pp. 30–37. No. CS-TR-1345 in Technical Report Series, Computing Science, Newcastle University (August 2012), http://www.cs.ncl.ac.uk/publications/trs/papers/1345.pdf

16. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Semantics of the VDM Real-Time Dialect. Tech. Rep. ECE-TR-13, Aarhus University (April 2013)
17. Mansfield, M., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 3. Tech. rep., INTO-CPS Deliverable, D3.6 (December 2017)
18. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: Bicarregui, J., Fitzgerald, J. (eds.) Proceedings of the Second VDM Workshop (September 2000)
19. Petridis, K., Clauß;, C.: Test of Basic Co-Simulation Algorithms Using FMI. In: Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015. pp. 865–872. No. 118, Fraunhofer IIS EAS, Zeunerstrasse 38, 01069 Dresden, Germany, Linköping University Electronic Press, Linköpings universitet (2015)
20. Thule, C., Gomes, C., Deantoni, J., Larsen, P.G., Brauer, J., Vangheluwe, H.: Towards the Verification of Hybrid Co-simulation Algorithms (2018), accepted for publication at CoSim-CPS-18
21. Thule, C., Lausdahl, K., Larsen, P.G., Meisl, G.: Maestro: The INTO-CPS Co-Simulation Orchestration Engine (2018), submitted to Simulation Modelling Practice and Theory
22. Verhoef, M.: On the Use of VDM++ for Specifying Real-Time Systems. Proc. First Overture workshop (November 2005)
23. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
24. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

All links were last followed on May 24rd, 2018.