Automatic generation of master algorithms for FMI 2.0 Co-Simulation

Fabio Cremona, Marten Lohstroh, Stavros Tripakis, Christopher Brooks and Edward A. Lee
UC Berkeley

11th Biennial Ptolemy Miniconference
Berkeley, CA
October 16, 2015
Outline

- Simulation and design of a Cyber-Physical System: Co-Simulation.
- FMI, a standard for the exchange of models and Co-simulation of CPSs.
  - Lacks, formalization and extensions.
- A Master Algorithm for Discrete Event and Continuous time dynamics.
- FIDE – An FMI Integrated Design Environment.
  - Designed to test our proposed extension to the standard.
Simulation of Cyber-Physical Systems

- Cyber-physical systems (CPS)
  - **Cyber**: computer-based systems, Discrete Events (DE) Model of Computation (MoC)
  - **Physical**: the model of the environment, Continuous Time system (CT) MoC

- Heterogeneous modeling
  - The design involves a wide breath of components and different area of expertise
  - Automata, state machines, transition systems, dataflow, discrete event systems, timed automata, ODEs, DAEs, PDEs, hybrid automata, ...
  - Different modeling paradigms, languages and tools for different components
Simulation of Cyber-Physical Systems

Cyber-Physical Systems: Cyber (digital, computer-based) + Physical

Typically heterogeneous modeling: state machines, discrete-event systems, differential equations, …

How these tools interact?

Low-level controllers
Simulink

Supervisory controllers
Rhapsody/
SysML

Physical dynamics
Modelica

Slide courtesy of Stavros Tripakis
Co-Simulation

- Co-Simulation permits simulating individual components using different simulation tools simultaneously and collaboratively.
- Co-simulation does not require agreement between tools on the semantics of models.
- Requirements for Co-Simulation.
  - **Semantics**: a model of computation to orchestrate the exchange of data and the advancement of time.
  - **Software engineering**: a standard interface for the components.
Functional Mock-up Interface (FMI)

- A tool independent standard for interoperable models
  - **Model exchange**: Model descriptions (FMUs) from one tool are interpreted and executed in another.
  - **Co-simulation**: Models created in one tool (FMUs) are executed within another tool.

---

Ptolemy Miniconference, October 16, 2015

Fabio Cremona, UC Berkeley
FMI 2.0 Co-Simulation

- Requires agreement on the semantics of the interface.
- FMU is a self-contained object: model + simulation engine provided by the design environment.
- The FMUs are orchestrated by a MA that deterministically exchange data and advance the simulation time.
Super-dense time? **NO!**

t \in \mathbb{R}

FMI uses **double** as data type for Real Numbers

- Timing precision is dependent on the magnitude of time, and operations on time incur complex quantization effects.

```c
double r = 0.8;
double k = 0.7;
k = k + 0.1;
printf("%f,%f,%d\n", r, k, r==k);
```

0.800000,0.800000,0
Limitations in FMI 2.0 Co-Simulation

- FMI defines five platform dependent data types for I/O and state variables:
  - `fmi2Real`, `fmi2Integer`, `fmi2Boolean`, `fmi2String`, `fmi2Char`, `fmi2Byte`
- These are continuous time variables.
- FMI lacks a notion of "absent" value, something that is essential for discrete-event and synchronous-reactive systems.
Limitations in FMI 2.0 Co-Simulation

- How to advance time in FMI?
- In Ptolemy II: fireAt() … it is pro-active! An actor asks to a director to be executed.
- In FMI: doStep() … the MA propose an advancement of time.
- An FMU can rejected or partially accept a step size!
- Roll-back?
Limitations in FMI 2.0 Co-Simulation

- What MoC for the Master Algorithm?
  - Not specified.
- Discrete Events?
  - No “absent”, no super-dense time
- Synchronous Reactive?
  - No “absent”, no super-dense time and no notion of “unknown”.
- Only sampled-data systems are well supported:
  - The step size can be fixed!
Some extensions to FMI 2.0 Co-simulation

Determinate Composition of FMUs for Co-Simulation

David Broman$^{1,2}$  Christopher Brooks$^3$  Lev Greenberg$^3$  Edward A. Lee$^1$
Michael Masin$^3$  Stavros Tripakis$^3$  Michael Wetter$^3$

$^1$University of California, Berkeley, USA  $^2$IBM
$^3$IBM
$^4$LBNL, Berkeley, CA, USA

ABSTRACT
In this paper, we explain how to achieve deterministic execution of FMUs (Functional Mockup Units) under the FMI (Functional Mockup Interface) standard. In particular, we model are either memoryless or implement one of rollback or step-size prediction. We show further that such a model can contain at most one “legacy” FMU that is not memoryless and provides neither rollback nor step-size prediction.

Requirements for Hybrid Cosimulation Standards

David Broman  Lev Greenberg  Edward A. Lee
KTH Royal Institute of  IBM Research – Haifa, Israel  UC Berkeley
Technology & UC Berkeley
Michael Masin  Stavros Tripakis  Michael Wetter
IBM Research – Haifa, Israel  UC Berkeley  Lawrence Berkeley National Laboratory

ABSTRACT
This paper defines a suite of requirements for future hybrid cosimulation standards, and specifically provides guidance for development of a hybrid cosimulation version of the XML schema for describing components. An FMU (Functional Mock-up Unit) is a component, typically exported from a modeling and simulation tool, that can be instantiated and used as part of a simulation in another modeling tool. To date, the emphasis of the standard has been on components.
Super-dense time

- FMU contract for DE semantics.

\[(A0) \text{ If } \text{doStep}_c(s, h) = (s', h') \text{ then } 0 \leq h' \leq h.\]
The need for **absent** to simulate DE dynamics.

- Each FMI data type has been enriched with the absent value:
  - \( V' = V \cup \{ \varepsilon \} \)
- This signal is always absent except at \( t = k*p, \ k \in \mathbb{N} \).
Efficient roll-back

- An efficient roll-back is the one you don’t have to do!
- FMU contracts for efficient rollback: predictable step size!

\[ \text{getMaxStepSize}_c : S_c \rightarrow \mathbb{R}_{\geq 0} \cup \{\infty\} \]

+ 

(A1) If \( \text{doStep}_c(s, h) = (s', h') \), then for any \( h'' \) where \( 0 \leq h'' \leq h' \), \( \text{doStep}_c(s, h'') = (s'', h'') \) for some \( s'' \).

= No need for roll-back
Some test components

- Test components and test compositions to evaluate the hybrid behavior.

---

**Modal Model with Discrete Control**

\[ y(t, n) = \begin{cases} \ a & \text{if } s(t, n) = 0 \\ \ b & \text{otherwise.} \end{cases} \]

\[ s(t, n) = \begin{cases} \ 0 & \text{if no such } i \text{ exists} \\ \ 1 & \text{if } s(d_i) = 0 \\ \ 0 & \text{if } s(d_i) = 1 \end{cases} \]

**Adder**

*Input* signals \( x_1 \) and \( x_2 \). *Output* signal \( y \).

For all \( \tau \in T \),

\[ y(\tau) = \begin{cases} \ x_1(\tau) + x_2(\tau) & \text{if } x_1(\tau) \neq \varepsilon \text{ and } x_2(\tau) \neq \varepsilon \\ x_1(\tau) & \text{if } x_1(\tau) \neq \varepsilon \text{ and } x_2(\tau) = \varepsilon \\ x_2(\tau) & \text{if } x_1(\tau) = \varepsilon \text{ and } x_2(\tau) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases} \]

**Periodic Piecewise Constant Signal Generator**

For all \( \tau \in T \),

\[ y(t, n) = \begin{cases} \ a & \text{if } kp < t < (k+1)p \text{ and } k \in \mathbb{N} \text{ is even;} \\ \ b & \text{if } kp < t < (k+1)p \text{ and } k \in \mathbb{N} \text{ is odd;} \\ \ b & \text{if } t \text{ is an odd multiple of } p \text{ and } n \geq 1; \\ \ a & \text{if } t \text{ is an even multiple of } p, t > 0, \ n = 0; \\ \ a & \text{otherwise.} \end{cases} \]
Open problem

- How to represent **time**?
  - This is still not solved.
  - What we know is that floating point numbers are not suitable to encode simultaneity of events.
FMI in a Nutshell

Notation:
- \( C \): set of FMU instances in a model
- \( c \in C \): FMU instance
- \( S_c \): set of states of FMU \( c \)
- \( U_c \): set of input ports of \( c \)
- \( Y_c \): set of output ports of \( c \)
- \( \mathbb{V} \): set of values that a port can take

API’s main functions:

- \( \text{init}_c : \mathbb{R}_{\geq 0} \rightarrow S_c \)  
  \( \text{init}_c(t) \mapsto s \)
- \( \text{set}_c : S_c \times U_c \times \mathbb{V} \rightarrow S_c \)  
  \( \text{set}_c(s, u, v) \mapsto s \)
- \( \text{get}_c : S_c \times Y_c \rightarrow \mathbb{V} \)  
  \( \text{get}_c(s, y) \mapsto v \)
- \( \text{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \rightarrow S_c \times \mathbb{R}_{\geq 0} \)  
  \( \text{doStep}_c(s, h) \mapsto (s', h') \)
What is an FMU?

- It is a black-box.
- It is a Mealy machine.
- A standard API to interact with the black box: set inputs, get outputs, advance state.

\[ \begin{array}{c}
\text{u (Input)} \quad \text{x (States)} \quad \text{y (Output)} \\
\end{array} \]

- \( \text{set}_c(s, u, v) \mapsto s \)
- \( \text{get}_c(s, y) \mapsto v \)
- \( \text{doStep}_c(s, h) \mapsto (s', h') \)
Models with feedback

- How to execute a model with feedback?

- Restriction to models with no cyclic dependencies.

Using I/O dependency information (c.f. lecture on synchronous systems).

FMI provides an (unfortunately optional) mechanism for an FMU to declare I/O dependencies.
- **get** known outputs $\rightarrow$ **set** dependent inputs $\rightarrow$ repeat (while respecting the dependencies), until all I/O ports are set $\rightarrow$ update (**doStep**) the states of all FMUs by calling **doSteps** (we’ll see how).

- FMI provides a mechanism for an FMU to declare I/O dependencies.
  - This allows the determination of a total order for I/O port update.
Updating the FMU States

- In what order should \texttt{doStep}_{FMU1} and \texttt{doStep}_{FMU2} be called?

- Suppose \texttt{doStep}_{FMU1} \texttt{doStep}_{FMU2}:
  - What if FM1 \texttt{accepts} \( h \) but FMU2 \texttt{rejects} it?

- Suppose \texttt{doStep}_{FMU2} \texttt{doStep}_{FMU1}:
  - What if FM2 \texttt{accepts} \( h \) but FMU1 \texttt{rejects} it?

- That's why we need an efficient mechanism for rollback!

Ptolemy Miniconference, October 16, 2015

Fabio Cremona, UC Berkeley
3.1 FMUs as Actors

In order to embed FMUs in a Ptolemy II model, we use a special actor called `FMUImport` that wraps the FMU and exports an actor interface. This pattern is essentially the same as the accessor described in [15]. All directors in Ptolemy II implement an actor abstract semantics (Figure 3) that divides the execution of a model up in three distinct phases: initialization, firing, and termination. For reasons beyond the scope of this paper, Ptolemy II splits up initialization and the firing in two sub-phases. The actor interface exposed by `FMUImport` (and any other actor in Ptolemy II) can be summarized as follows:

- `preinitialize()`, executes higher order functions that build model structure. It is invoked exactly once during the execution of the model and is executed before any static analysis;
- `initialize()`, initializes the variables for execution;
- `fire()`, lets the actor reads inputs, perform computation, and generate outputs;
- `postfire()`, updates the state of the actor;
- `wrapup()`, terminates the simulation.

The master algorithm from [6], illustrated by Figure 4, shows a similar division of phases. Indeed, `FMUImport` establishes a mapping between the two interfaces. The pseudo code in Figure 5 explains how.

In `FMUImport`, `preinitialize()` is in charge of linking and instantiating the FMU library. First, the method checks whether or not the FMU is already compiled, and if needed, it compiles the FMU as shared library. Subsequently, it instantiates the FMU and parses the `ModelDescription` XML-schema in order to instantiate corresponding actor ports and parameters accordingly. By double-clicking the icon of `FMUImport` the user can customize the configuration of the FMU. If changes are made, the `initialize()` method will override the default values with the new user-defined values. The `fire()` method performs the computation of a time step. It sets the inputs of the FMU using `fmi2SetXXX(...)`, then it invokes `fmi2GetXXX(...)` to trigger the computation of new outputs and retrieve them, and finally, `fmi2DoStep(...)` is invoked to advance time by a given delta. Note that the FMU may reject the suggested step size and return a smaller delta, 0. In this case, the state of the actor will not be updated. Instead, `postfire()` requests a new firing from the director with the adjusted step size. This sequence continues until the end of the simulation is reached. When the simulation terminates, `wrapup()` deallocates the current instance of the FMU.

3.2 Code Generation

Building on Ptolemy II’s “cg” framework (see Section 2.3.1), to implement C-code generation for FMI, we extend the class `ProceduralCodeGenerator` to tailor it to the specifics of FMI and load a template file that outlines the master code. The structure of this template file corresponds to the layout of Figure 4. To provide an indication of the implementation effort, this class, `FMIMACodeGenerator`, only counts 272 lines of code. The bulk of the implementation is in FMI-specific adapters for the Director (333 lines) and `TypedCompositeActor` (266 lines) — classes that constitute the basic primitives of a Ptolemy II model. The code generator generates glue code between the FMUs and then creates a “.c” file that makes calls to a library based on the Qtronic’s FMU SDK library containing functionality to link the FMUs and parse their model description schema. For a model with 8 FMUs, the generated file with the co-simulation algorithm for FMI is 623 lines long.
FIDE – An FMI Integrated Design Environment

- Imports FMUs as FMU-actors with input and output ports.
- Arrange and interconnect FMU-actors through a graphical user interface.
- Co-simulate a composition of FMUs using an implementation of the MA. The MA is generated as C-code that can be compiled and executed outside Ptolemy II with benefits in performance and portability.
Super-dense time
- Multiple iterations at the same time synchronization point
- “absent”
  - The FMI API has been extended to introduce

```c
fmi2Status fmiSetHybridXXX (fmi2Component c, const fmi2ValueReference vr[],
                                int nvr, const fmi2XXX value[], fmi2SignalStatus status[]);

fmi2Status fmi2GetHybridXXX (fmi2Component c, const fmi2ValueReference vr[],
                                 int nvr, fmi2XXX value[], fmi2SignalStatus status[]);

typedef enum {
    PRESENT,
    ABSENT,
    UNKNOWN,
    PRESENT_THEN_ABSENT
} fmi2SignalStatus;
```

Though for future extensions
Example

Zero-Delay Feedback

Constant

\[
y(t, n) = 1
\]

Integrator with Reset

Check Values

Adder

Zero-Crossing Detector

Discrete Microstep Delay

Ptolemy Miniconference, October 16, 2015  Fabio Cremona, UC Berkeley
Thanks!

11th Biennial Ptolemy Miniconference

Berkeley, CA
October 16, 2015