FIDE – An FMI Integrated Development Environment*

Fabio Cremona
University of California, Berkeley, USA, and Scuola Superiore Sant'Anna, Italy
f.cremona@eecs.berkeley.edu

Marten Lohstroh
University of California, Berkeley, USA
marten@eecs.berkeley.edu

Stavros Tripakis
University of California, Berkeley, USA, and Aalto University, Finland
stavros@eecs.berkeley.edu

Christopher Brooks
University of California, Berkeley, USA
cxh@eecs.berkeley.edu

Edward A. Lee
University of California, Berkeley, USA
eal@eecs.berkeley.edu

ABSTRACT

This paper presents FIDE, an Integrated Development Environment (IDE) for building applications using Functional Mock-up Units (FMUs) that implement the standardized Functional Mock-up Interface (FMI). FIDE is based on the actor-oriented Ptolemy II framework and leverages its graphical user interface, simulation engine, and code generation feature to let a user arrange a collection of FMUs and compile them into a portable and embeddable executable that efficiently co-simulates the ensemble. The FMUs are orchestrated by a well-vetted implementation of a master algorithm (MA) that deterministically combines discrete and continuous-time dynamics. The ability to handle these interactions correctly hinges on the implementation of extensions to the FMI 2.0 standard. We explain the extensions, outline the architecture of FIDE, and show its use on a particularly challenging example that cannot be handled without the proposed extensions to FMI 2.0 for co-simulation.

CCS Concepts

•**Computing methodologies** → **Discrete-event simulation; Continuous simulation;** *Simulation languages; Simulation tools;*

Keywords

Functional Mock-up Interface (FMI); Simulation; Co-simulation; Ptolemy II; Master Algorithm

*Acknowledgments: This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies) and by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, #1446619 (Mathematical Theory of CPS) and the following companies: Denso, National Instruments, and Toyota. This work has also been supported by the Academy of Finland and by NSF awards #1329759 and #1139138.

SAC'16, April 4-8, 2016, Pisa, Italy

ACM 978-1-4503-3739-7/16/04…$15.00
http://dx.doi.org/10.1145/2851613.2851677

1. INTRODUCTION

Cyber-Physical Systems (CPS) are characterized by the conjunction of a Cyber component (a controller) and a Physical component (a plant). Modeling the interaction between these components often spans different paradigms, including continuous time, modal models, and discrete events. Design and simulation of a CPS usually involves multiple teams working simultaneously on different aspects of the system. The design of a controller typically requires different models and tools than are used in the design of a physical plant. Even modeling the physical system alone may involve a breadth of components, each of which may rely on a separate area of expertise (e.g., electrical, mechanical, chemical, thermal, etc.) and may require a separate toolkit.

Co-simulation permits a model of a system to comprise components that are simulated using different tools. The communication between components (slaves) is orchestrated by an external program (a master) that implements what is often referred to as a **master algorithm** (MA). This algorithm generally involves querying components for outputs, setting component's inputs, and advancing model time.

FMI (Functional Mock-up Interface) is a standard initiated by Daimler AG within the ITEA2 MODELISAR project [3, 4, 20], and is now maintained by the Modelica Association. It has been designed to enable the exchange and interoperation of model components or subsystems designed with different modeling tools. In FMI, a component is called an **FMU** (Functional Mock-up Unit). The FMI standard describes two different modes of operation: **FMI for model exchange** (FMI-ME) and **FMI for co-simulation** (FMI-CS).

In FMI-ME, an FMU only declares a set of variables, equations, and optional data such as parameter tables or user interface features. This mode provides a common format for the exchange of components across different simulation tools. For instance, one can design a component in Modelica [19], export it as an FMU, and then import and use it in Simulink[1]. An FMU for model exchange is a "passive" component, meaning that the host environment must solve the equations provided by the FMU. In FMI-CS, on the other hand, an FMU is a self-contained object that besides the model description also includes a simulation engine provided by the

[1] www.mathworks.com/products/simulink/

design environment in which it was created. For example, a co-simulation FMU may provide the functionality of an executable Simulink model.

In either mode, the execution of a collection of FMUs is specified to be orchestrated according to a master algorithm. In co-simulation mode this only involves coordinating the exchange of data between interconnected FMUs and the advancement of time, whereas in model-exchange mode the MA must also solve the equations provided by each FMU. The FMI standard, however, leaves the MA largely unspecified and describes only on the interface contract between FMU and MA.

This paper introduces an **Integrated Design Environment** (IDE) for FMI called **FIDE** (FMI-IDE). FIDE allows the modeling and design of systems through co-simulation of FMUs. FIDE is part of the **Ptolemy II** framework [11, 22] — an actor-oriented modeling suite for CPS — and it leverages Ptolemy II's graphical user interface (GUI) and code generation package in order to generate a pure C co-simulation implementation that is potentially very efficient and, like an FMU, can run on a wide variety of platforms.

The paper is organized as follows. In the remainder of this section we discuss related work. In Section 2 we discuss the API specified by the FMI standard for co-simulation and we describe the master algorithm we use to orchestrate co-simulation of FMUs. In Section 3 we describe the architecture of our IDE. Section 4 demonstrates a practical example of its use. Finally, in Section 5 we provide our conclusions and a discuss possible avenues to be explored as future work.

1.1 Related Work

Co-simulation has a wider application than merely integration of design artifacts that are crafted in different tools. Metropolis [1], for instance, uses the concept of co-simulation to define a meta-model that maps functionality to its implementation on architectural elements. The design flow enabled by Metropolis permits analysis and formal verification to determine how well an implementation satisfies a set of abstract requirements at each step of refinement towards that implementation. This is achieved by simulating functional models and tightly coupling them with a simulation of the hardware that the functionality is mapped onto. This idea is leveraged by numerous other tools and various application spaces. For instance, the TRes framework [10], models through co-simulation how the performance of a functional model is impacted by architectural elements such as scheduling and the execution time delays. In a completely different domain, the Building Control Virtual Test Bed [26] is used to model building-heat transfer and HVAC system dynamics in order to simulate the effects of control algorithms and building parameters.

FMI is an attempt to define a unifying kernel for computer-aided design artifacts and make them interoperable through a standardized interface. This standard is evolving — at time of this writing, the latest version is FMI 2.0. Information about the standard with a list of compatible tools, the original specification of the standard, and a list of related publications can be found on the FMI website[2].

[2] https://www.fmi-standard.org

The impetus for the development of the presented IDE was to facilitate prototyping and experimentation with extensions of the FMI standard. Broman et al., in [6], proposed extensions for the support of co-simulation of FMUs where inputs and outputs may be *hybrid* signals, i.e., mixtures of discrete events and continuous time signals. Importantly, Broman et al. explain the conditions under which the simulation algorithm produces a deterministic execution trace and show that each integration step in their algorithm terminates. We implemented the MA described in [6], and using FIDE, successfully generated code that co-simulates several FMUs. Whereas Broman et al. provide a neat formal framework, FIDE complements their effort by providing a managed build process that delivers runnable code amenable to regression tests. The example we highlight in Section 4 is drawn from [7], a paper that provides a suite of requirements and test cases for future hybrid co-simulation standards.

It should be noted that, prior to the publication of this paper, Bogomolov et al. [5] have already reported success leveraging our toolchain as a host environment to co-simulate FMUs exported by two state-of-the-art modeling and verification tools for hybrid systems, SPACEEX [12] and UPPAAL [15].

Of course, other efforts have been made to develop tools for FMI. Although this section by no means provides a comprehensive survey, the following tools show some overlap with FIDE in form and function. QTronic developed FMU SDK[3], which is a freely available FMU development kit that allows basic usage of FMI and provides functionality to test single FMUs. It supports both FMI-ME and FMI-CS. Yet, the FMU SDK does not actually "co-simulate" as it can only execute one single FMU in isolation. The FMI Toolbox for MATLAB/Simulink from Dymola[4] offers import and export functionality of FMUs in Simulink for both model exchange and co-simulation. This allows FMUs to interoperate through the Simulink execution engine. Bastian et al. in [2] discuss a MA implementation for FMI-CS that is designed to run on multiple platforms (MS Windows, Linux, Sun Solaris) and uses OpenMP to parallelize the execution of FMUs. Their MA is limited to the use of a fixed communication step size and it requires that the outputs of FMUs do not depend on the current output of other FMUs. Neema et al. in [21] take a different approach and use a general purpose architecture for distributed computer simulation systems called High-Level Architecture (HLA) [14] as a MA. DACCOSIM [13] is a co-simulator developed by the RISEGrid institute that can co-simulate FMUs in parallel on different nodes in a cluster. It features a GUI to compose FMUs and map segments of the graph to different cluster nodes. A code generator outputs Java code that wraps the FMUs in the appropriate master code. Parallel execution is coordinated using Parallel Python.

Although DACCOSIM claims to handle mixtures of continuous and discrete signals, their implementation is more an approximation of event handling. Following the definition in [7], events are instantaneous. Instead, in DACCOSIM, events happen during a time *interval*, not at a precise time instant.

Unlike FIDE, all aforementioned tools operate within the

[3] www.qtronic.de/en/fmusdk.html
[4] www.modelon.com/products/fmi-toolbox-for-matlab/

bounds of the FMU-CS 2.0 standard and are therefore unable to faithfully co-simulate systems that feature both continuous and discrete dynamics.

Finally, it is in order to mention Tripakis's recent work [24] because it discusses how to translate different modeling formalisms (untimed and timed state machines, discrete events, and dataflow models) into FMI 2.0 co-simulation. Ptolemy II, the framework that FIDE builds on, features each of these formalisms. The relationships between these formalism and the semantics of a master algorithm for co-simulation are crucial for answering the question as to whether certain useful semantic properties can be preserved when a composition of FMUs in Ptolemy II is translated into FMI co-simulation. Future work outlined in Section 5 involves further elaboration on this question.

2. BACKGROUND

2.1 FMI for Co-Simulation

The FMI standard defines an **application programming interface** (API) that all components conforming to the standard must implement. An FMU is implemented as a combination of XML-files and C code (either compiled, as a DLL/shared library, or as source code). In an industrial setting, the possibility to share FMUs as shared libraries is particularly useful to protect **intellectual property** (IP) as the FMU supplier can keep the FMU source code hidden. The XML description file, following an FMI description schema, defines variables and their attributes such as name, unit, initial value, etc. The XML file also contains information about the structure of the model. The *ModelStructure* section in the FMI description schema contains information about dependencies between derivatives, outputs, and inputs. As shown in Section 2.2, I/O-dependency information is used to determine the I/O update sequence.

Abstractly speaking, an FMU can be seen as a timed Mealy machine [24], consisting of a set of input variables (**ports**), a set of output variables/ports, and collection of internally kept **state**, inaccessible from the outside world. The outside world interacts with the FMU only by means of its API, which consists primarily of the following methods (as in [6] we use shorter names for the methods than the ones defined in the FMI standard, for brevity):

- `init`, which initializes the FMU given a time instant;
- `set`, which assigns a given input variable a given value;
- `get`, which returns the value of a given output variable;
- `doStep`, which attempts to perform a simulation step on the FMU, given a time step Δt (a non-negative real number). This corresponds to the machine making a transition, i.e., updating its internal state. Since the machine is timed, the transition is also timed, i.e., it depends on the given delay Δt and corresponds to advancing time by a certain amount. For reasons having to do with numerical integration and other simulation issues that are beyond the scope of this paper, a call to `doStep` may *succeed*, in which case the FMU indeed makes a step of Δt (we say that the FMU *accepts* Δt), or it may *fail*, in which case the FMU makes a smaller step of $\Delta t' < \Delta t$ (we say that the FMU *rejects* Δt). The success or failure status is returned by the method, and $\Delta t'$ is returned in case of failure.

It should be noted that version 2.0 of the FMI standard forbids time steps of size 0. However, in this paper we shall not make this restrictive assumption, since zero-time steps are essential to modeling "cyber" phenomena (discrete transitions, happening instantaneously) and hybrid systems (see [7, 24, 6] for more extensive discussions on this topic).

Several additional methods apart from the above are defined in the FMI standard, including methods to save and restore the internal state of an FMU. These methods are *optional* (meaning that an FMU is not *required* to implement them), but as we will explain next, they are essential for the operation of a properly functioning master algorithm.

2.2 Deterministic Master Algorithm

The FMI standard defines the API that FMUs must implement, but does not define the master algorithm (i.e., the co-simulation algorithm). The problem of devising a master algorithm with "good" characteristics was studied in [6]. That work proposed two master algorithms which achieve **determinism**, meaning that the results of the simulation do not depend on arbitrary factors such as FMU names, creation times, or any other arbitrary order in which the FMUs might be chosen during the various iterations used in the algorithms. Our paper relies on these algorithms, so we briefly describe their basic scheme here.

The input to the algorithm is a model consisting of a set of interconnected FMUs, as in the diagram shown in Figure 1. The deterministic master algorithm starts by initializing the state of all FMUs to time 0 (the start time of the simulation). Then the algorithm proceeds with a sequence of simulation steps, up to a given number of steps, or until a given simulation time is reached. Each simulation step consists of two phases. In the first phase, the values of all input/output ports are propagated through the model. In the second phase, a time step Δt is chosen and all FMUs perform a step with the chosen Δt. The current simulation step is then over, and a new one can begin. Let us next describe these two phases in more detail.

The first phase consists of a sequence of calls to `get`s and `set`s on the FMUs of the model, until the values of all output ports and input ports of the model are up to date. One potential problem here is the presence of feedback loops in the model, which may result in cyclic dependencies. The method proposed in [6] solves this problem by relying on the input/output dependency information contained in each FMU, to ensure that the model is essentially acyclic, in spite of feedback loops.

For example, consider the model of Figure 1. If in every FMU of this model, all outputs depend on all inputs, then the model has cyclic dependencies and cannot be handled by the method of [6]. This approach is fully compliant with FMI-CS. Techniques to solve cyclic dependencies, such as the Newton-Raphson method and the successive substitution method [9], require multiple iterations among the model equations, i.e., multiple calls of `get` and `set` without advancing the model time. However, according to FMI-CS 2.0, cyclic dependencies

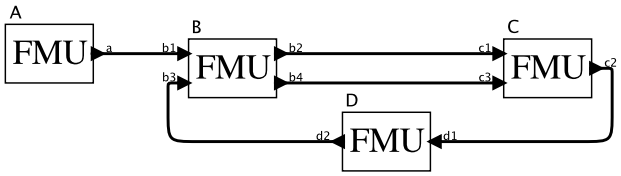

Figure 1: A model consisting of FMUs connected in a block diagram.

can be handled *only* in "Initialization Mode" (before the simulation starts); it is not allowed to iterate among the model equations to solve cyclic dependencies in "Step Mode" (during the simulation).

Now consider the model in Figure 1 and assume that outputs $b2$ and $b4$ of FMU B do not depend on input $b3$. Indeed, the model is *acyclic*. The input/output values in this model can be propagated in the following order: `get` the value of port a, `set` the value of $b1$, `get` the values of $b2$ and $b4$ (note that here we use the fact that $b2$ and $b4$ only depend on $b1$, whose value is already known), `set` $c1$ and $c3$, `get` $c2$, `set` $d1$, `get` $d2$, and finally, `set` $b3$. At this point, all port values are set, and the first phase is over.

In the second phase, `doStep` needs to be called on each FMU in the model. Here, the problem is to choose the right time step Δt, such that it is accepted by all FMUs (otherwise, the FMUs may end up in a desynchronized state, with some having advanced their local time further than others — this can obviously lead to wrong simulation results).

The method of [6] (Figure 2) solves this problem as follows. First, the internal states of all FMUs in the model are saved. Then, `doStep` is called on all FMUs with a default time step, $\Delta t_{\max}$. If all FMUs accept it, then the step is successful and the algorithm can proceed to the next simulation step. Otherwise, the time step is set to the *minimum* of all Δt's returned by the calls to `doStep` that failed. This minimum, call it $\Delta t_{\min}$, corresponds to the maximum time amount which is guaranteed to be accepted by all FMUs (the assumption here is that if an FMU was able to advance by some $\Delta t > \Delta t_{\min}$, then it will also be able to advance by $\Delta t_{\min}$). Then, the saved states of all FMUs are restored (*rollback*) and `doStep` is again called on all FMUs, with time step $\Delta t_{\min}$. This is guaranteed to succeed for all FMUs, and the algorithm can proceed to the next simulation step.

2.3 Ptolemy II

Ptolemy II is an open-source software framework for actor-oriented modeling, analysis and design of heterogeneous systems. **Actors** are components that execute concurrently and share data with each other by sending messages via ports. Ptolemy models are hierarchical, and on each level of hierarchy the interaction patterns between actors are constrained by the rules imposed by a **director**. These interaction rules constitute a **Model of Computation** (MoC) [22].

2.3.1 The "cg" Code Generator

The Ptolemy II code generator is a template-based system that generates code in C [25], Java [23] and other languages. The

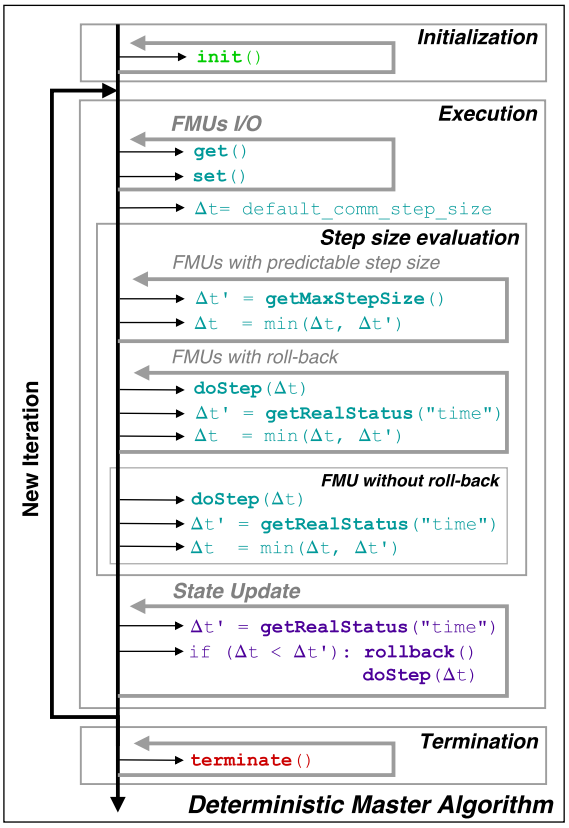

Figure 2: Deterministic Master Algorithm proposed in [6].

code generator itself is implemented in Java. The "cg" system traverses a Ptolemy II model, analyzes the types and then reads **template files** that correspond to key components such as directors, actors, and parameters. The template files are typically language-dependent, though some infrastructure can be shared between similar languages. A small language defines substitutions that occur to support actor activities such as reading and writing of ports and parameters. The template file for the director defines the glue code that updates the values of ports and then executes the actors. The order in which ports are updated and the actors executed define the semantics of the director that is preserved during code generation. After stitching together the templates, files are generated and compiled.

3. FIDE

FIDE provides the following key abilities:

- Import FMUs as FMU-actors;
- Arrange and interconnect FMU-actors through a GUI;
- Co-simulate a composition of FMUs using an implementation of the MA described in [6]. The MA is generated as C code that can be compiled and executed outside Ptolemy II with benefits in performance and portability.

The design flow of FIDE is outlined in Figure 3. First, the user selects FMUs from the file system and imports them. The

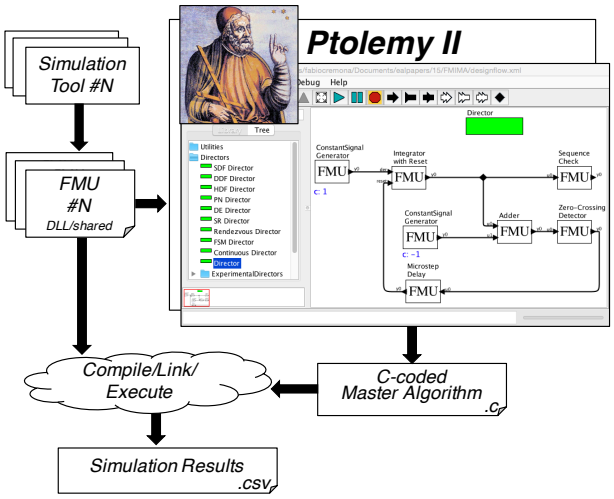

Figure 3: FIDE Design flow.

FMUs appear as actors in a diagram, and the user can wire them together. Different directors (MoCs) are available to simulate the ensemble using the Ptolemy II execution engine. FMUs are originally designed to only work with the **Continuous Time** (CT) [8] director, but FMUs compliant with the recommendations in [6] can be simulated using the **Discrete Events** (DE) [17] director as well. The next step is to generate C code based on the arrangement of FMUs in the model. As a result of this process, a folder containing a ".c" file with an embedded co-simulation execution engine and a set of support files is generated. Compilation can be done on any machine, and the FMUs are loaded dynamically at run time. Finally, execution of the obtained binary results in a ".csv" file containing a trace of the simulation, comprising the values of all interface variables of all FMUs at each synchronization point along the simulation time line.

The following few paragraphs discuss implementation details of the toolchain.

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 very similar to the *accessor*, described in [16]. All directors in Ptolemy II implement an actor abstract semantics (Figure 4) that divides the execution of a model up into three distinct phases: *initialization*, *execution*, 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;

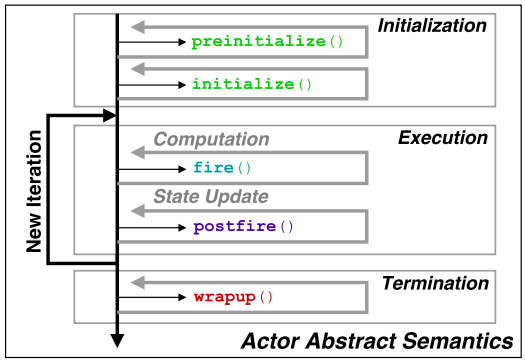

Figure 4: The actor abstract semantics in Ptolemy II.

- `wrapup`, terminates the simulation.

The master algorithm from [6], illustrated by Figure 2, shows a similar division of phases. Indeed, `FMUImport` establishes a mapping between the two interfaces. The code segments in cyan are executed in `fire` and the code segments in magenta are executed in `postfire`. The *initialize* and *termination* phase colored green and red, respectively, are trivially mapped to their actor-oriented counterpart.

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. By double-clicking on 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 `set`, then it invokes `get` to trigger the computation of new outputs and retrieve them, and finally, `doStep` is invoked to advance time by a given delta. Note that the FMU may reject the suggested step size and return a time step $\Delta t' < \Delta t$. 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 $\Delta t'$. 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 2. The bulk of the implementation is in FMI-specific adapters for the `Director` and `TypedCompositeActor` 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.

3.3 The Master Code

Next, we discuss the specifics of the template file that implements the master algorithm.

Initialization.

The initialization code instantiates and configures the FMUs that appear in the input block diagram that the user composes in Ptolemy II. This involves parsing the model description schema of each FMU. In this process, key information is extracted, including the FMU capability flags. Specifically, the flags `canGetAndSetFMUState` and `canGetMaxStepSize` are required for the execution of the simulation algorithm (see Section 2.2 and Broman et al. [6]). If all FMUs are found to be compatible, they are instantiated, and the FMU parameters are set to the values supplied in the Ptolemy II model.

This segment of the template also contains code that determines the execution order among FMUs. It generates a directed graph based on the input model (the block diagram of FMUs). The vertices in this graph, however, are input and output ports — *not actors*. The arcs represent connections between ports (relations). We analyze the graph and reject the model if the graph is not acyclic (if the graph contains a cycle then the model features an algebraic loop. The **directed acyclic graph** (DAG) is used to derive a total order that describes the evaluation order for the I/O ports [6].

```
typedef struct {
    FMU* sourceFMU;
    FMU* sinkFMU;
    fmi2ValueReference sourcePort;
    fmi2ValueReference sinkPort;
    fmi2ValueType sourceType;
    fmi2ValueType sinkType;
} portConnection;
```

Figure 5: The port connection structure.

For each arc in the DAG we keep a `portConnection` struct, described in Figure 5. Here we store a pointer to the *source* and *sink* FMU at the end-points of a connection. The `sourcePort` and `sinkPort` fields contain value references that indicate the particular output port and input port associated with the connection (in FMI, the value reference unequivocally identifies a variable inside the FMU). The data types of the source and sink port are stored in the `sourceType` and `sinkType` fields. Knowledge of the data type of a signal is critical in FMI because the FMI standard defines different API methods to access the different data types defined in the standard. The `fmi2ValueType` field is defined as an enumeration where the values correspond to the data types defined in FMI. All instances of `PortConnection` are ordered in accordance with the determined evaluation order and stored in an array.

Simulation.

The simulation loop contains a hard-coded version of the co-simulation algorithm described in Section 2.2 and in [6]. The input parameters to the simulation algorithm are an array of pointers to all the FMU instances, the connection list generated during initialization, and a default value for the step size $\Delta_{default}$. A `while`-loop iterates the simulation until a variable containing the current simulation time is greater than the final simulation time or an error occurs. At each iteration, the algorithm updates all the I/O ports, negotiates the maximum step size that can be accepted by all FMUs, and advances the state of all the FMUs. Finally it saves the variable containing the current time and the FMU's state in a ".csv" file.

Termination.

This section contains hard-coded `C` functions to terminate and deallocate the FMUs.

3.4 Extensions to the FMI Standard

Hybrid co-simulation may involve mixtures of discrete events and continuous time signals. However, FMI-CS has been designed only with continuous time signals in mind: signals are present at *every* time instant $\tau \in T$ ($T = \mathbb{R}_+$ represents time), they can never be absent ($x(\tau) \neq \varepsilon$). A discrete event signal [18] is present only at *some* time instants $\tau \in \mathcal{D} \subset T$, where $\mathcal{D}$ is a discrete set. To rigorously model discrete event signals, we adopted the superdense model of time [22], as already supported by FMI-ME. Superdense time $\tau \in T$ (with $T = \mathbb{R}_+ \times \mathbb{N}$) is a two-tuple $\tau = (t, n)$. The real number t represents a time in the usual Newtonian sense, while n represents a **microstep**, the index of an iteration at the same communication point (at the same Newtonian time t). With superdense time, we can model value changes of signals happening at the *same* Newtonian time t, by only advancing the microstep index. In FMI-CS, the step size for `doStep` is constrained to be $\Delta_t > 0$. Every time a state update is performed (`doStep` is invoked), the standard imposes that time advances in the Newtonian sense. To support hybrid co-simulation, we relax this particular constraint, and we allow $\Delta_t = 0$.

The handling of discrete event signals requires an extension of the current FMI API as it does not support absent values for `get` and `set`. We define new API functions to overcome this problem:

```
fmi2Status fmi2SetHybridXXX (fmi2Component c,
    const fmi2ValueReference vr,
    size_t nvr, const fmi2XXX value[],
    const fmi2SignalStatus flag[])

fmi2Status fmi2GetHybridXXX(fmi2Component c,
    const fmi2ValueReference vr[],
    size_t nvr, fmi2XXX value[],
    fmi2SignalStatus flag[])
```

Figure 6: FMI API extensions.

These functions extend the original `get` and `set` by introducing an additional argument, `fmi2SignalStatus flag[]`, that is defined as an enumeration (`typedef enum {present, absent} fmi2SignalStatus;`). If (`flag == present`), the signal is present, or otherwise (`flag == absent`) the signal is absent. The enumerate can be extended to encode other "special" values such as *unknown* ($\perp$), which plays a critical role in fixed-point computations.

While these extensions are supported by our MA, we also designed and implemented a library of FMUs that makes use of them. Finally, even though our implementation does not strictly obey the standard, our implementation is backward compatible with FMUs designed for FMI-CS, which do not rely on the "hybrid" capability.

4. EXAMPLE

We tested the capability of our tool, by implementing the components (FMUs) and the use cases as described in [7] that exemplify the need for the extensions to the current FMI-CS 2.0 standard which are described in section 3.4. From these use cases, we picked the one that is most challenging to present here. The model is illustrated in Figure 7. Despite the deceptively small size of this example, it features intricacies that FMI-CS 2.0 compliant tools will not be able to cope with.

First of all, there is a `Zero-CrossingDetector` present. In simulation environments like Simulink and Ptolemy II, a Zero-CrossingDetector relies on **backtracking** for a precise identification of a time instant at which a zero crossing occurred. Instead of progressing the simulation in small time steps in order to account for a possible zero-crossing — these are typically sparse events — the simulation engine adjusts the step size retroactively, when a zero crossing was detected. The algorithm from [6] does not support backtracking. Instead, we overcome this problem by propagating the derivatives from the `IntegratorWithReset` through the `Adder` to the `Zero-CrossingDetector`. Based on its current input and derivative(s), the `Zero-CrossingDetector` is able to compute the exact time at which the next zero crossing will occur, and compute its step size accordingly.

Upon detecting a zero crossing, the `Zero-CrossingDetector` also generates a discrete event, which causes the other components to iterate in superdense time. The generation of this event follows the description provided in [7]; at a single Newtonian time instant the output changes from $\varepsilon \to 0 \to \varepsilon$. This is achieved by having the `Zero-CrossingDetector` propose a step size $\Delta_t = 0$. All the components in the model react synchronously to the event. This semantics enables the `IntegratorWithReset` to catch the reset event and instantaneously react by producing an output $y_0 = 0$.

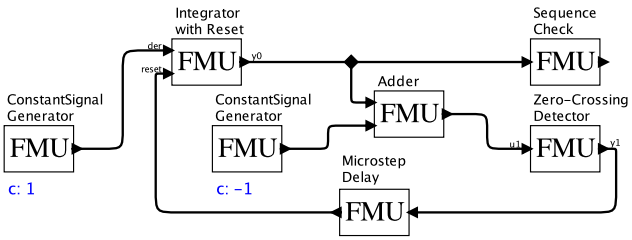

Figure 7: Test model using a Zero-Crossing Detector in a feedback loop.

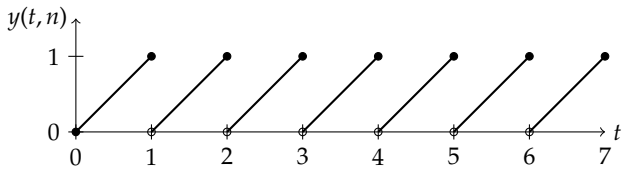

Figure 8: Output obtained from FIDE after compiling and running the model in Figure 7.

The graph in Figure 8 shows the output from the `IntegratorWithReset` from the model in Figure 7 for a simulation length of 7 *seconds*. The `IntegratorWithReset` integrates the input signal, a constant signal with value $c = 1$, and produces a "ramp" with a slope of 1. The initial condition for the integrator is $y_0 = 0$. From time $\tau = (0, 0)$, the integrator starts producing a ramp-signal with the initial value $y_0 = 0$. When the value of the signal reaches $y_0 = 1$, a zero crossing is detected by the `Zero-CrossingDetector`, and, at the same Newtonian time, the state of `IntegratorWithReset` is reset to $y_0 = 0$. More precisely, the output of the `IntegratorWithReset` is $y_0 = 1$ at time $\tau = (1, 0)$, $y_0 = 1$ at time $\tau = (1, 1)$, and $y_0 = 0$ at time $\tau = (1, 2)$. At time $\tau = (1, 0)$, the input of the `Zero-CrossingDetector` becomes 0, but the resulting event is only present at time $\tau = (1, 1)$. The reset of the integrator actually occurs only at microstep two, because the event associated with the zero crossing detection is delayed by one additional microstep due to the `MicroStepDelay`. The role of `MicrostepDelay` is to break the causality loop due to the direct dependency of y_0 on the *reset* input in the `IntegratorWithReset`. After $\tau = (1, 2)$ the sequence repeats itself until the end of the simulation.

5. CONCLUSION AND FUTURE WORK

FIDE provides a toolchain for conveniently mocking-up applications using co-simulated FMUs. A graphical user interface facilitates the arrangement of FMUs, after which an automated process compiles the ensemble into a stand-alone application. This approach bypasses the Ptolemy II execution engine, which gives more flexibility for the experimentation with different master algorithm implementations and possible extentions to the API as it is defined by the FMI standard.

The "default" master code that coordinates the co-simulation in FIDE features the API extensions from [6]. The extensions allow us to co-simulate models that combine continuous and discrete dynamics, something which other tools, obeying the FMI-CS 2.0 stadard, are unable to do. The master is modeled after the Continuous Time director in Ptolemy II, but we have no proof that their semantics are equivalent. A sound mapping from the Continuous Time director to our master code may be possible on the basis of [24].

We plan to develop a "hybrid" version of the FMUImport actor in order to co-simulate FMUs using the Ptolemy II execution engine. As we continue to build our library of regression tests, we expect to see that our FMI co-simulation master will produce the same simulation results as the Ptolemy II CT director. Preliminary results indicate that the code-generated co-simulations have a significant performance advantage over the Ptolemy II execution engine.

As part of the IDE, we supply a library of FMUs and implementations of the models discussed in [7]. FIDE is embedded in Ptolemy II framework, which is downloadable from http://ptolemy.eecs.berkeley.edu.

6. ACKNOWLEDGMENTS

We thank Michael Wetter from Lawrence Berkeley National Laboratory for his contributions to the implementation of the `FMUImport` actor and Fabian Stahnke from Technische Universität München who has contributed to the code generation framework for FMI. The "cg" framework in Ptolemy

II has seen many contributors, among whom, we acknowledge Man-kit (Jackie) Leung, Gang Zhou, Ye Zhou, and Bert Rodiers.

7. REFERENCES

[1] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe. Metropolis: an Integrated Electronic System Design Environment. *Computer*, 36(4), 2003.

[2] J. Bastian, C. Clauss, S. Wolf, and P. Schneider. Master for Co-Simulation Using FMI. In *8th Modelica Conference*, pages 115–120, 2011.

[3] T. Blochwitz, M. Otter, et al. The Functional Mockup Interface for Tool Independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, 2011.

[4] T. Blochwitz, M. Otter, et al. Functional Mock-up Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models. In *Proceedings of the 9th International Modelica Conference*, 2012.

[5] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikucionis, T. Strump, and S. Tripakis. Co-Simulation of Hybrid Systems with SpaceEx and Uppaal. In *Proceedings of the 11th International Modelica Conference*, 2015.

[6] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2013)*. IEEE, 2013.

[7] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Requirements for Hybrid Cosimulation Standards. In *Proceedings of 18th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 179–188. ACM, 2015.

[8] J. Cardoso, E. Lee, J. Liu, and H. Zheng. Continuous-time models. In C. Ptolemaeus, editor, *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[9] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, 2006.

[10] F. Cremona, M. Morelli, and M. Di Natale. TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1940–1947, New York, NY, USA, 2015. ACM.

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity—the Ptolemy Approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[12] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

[13] V. Galtier, S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui, and G. Plessis. FMI-Based Distributed Multi-Simulation with DACCOSIM. In *Symposium on Theory of Modeling and Simulation, TMS'15*, pages 804–811, Alexandria, VA, USA, April 2015.

[14] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.

[15] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2), 1997.

[16] E. Latronico, E. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A Vision of Swarmlets. *Internet Computing, IEEE*, 19(2):20–28, Mar 2015.

[17] E. A. Lee, J. Liu, L. Muliadi, and H. Zheng. Dicrete-event models. In C. Ptolemaeus, editor, *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[18] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, 2005. Springer-Verlag.

[19] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3 Revision 1*, 2014. Available from: http://www.modelica.org.

[20] MODELISAR Consortium and Modelica Association. Functional Mock-up Interface for Co-Simulation, October 12, 2010. Version 1.0, Retrieved from https://www.fmi-standard.org.

[21] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar. Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In *10th International Modelica Conference*, pages 10–12, 2014.

[22] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, 2014.

[23] M. Schoeberl, C. Brooks, and E. A. Lee. Code Generation for Embedded Java with Ptolemy. In *Proceedings of the 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2010)*, October 2010.

[24] S. Tripakis. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation – SAMOS XV*, 2015.

[25] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in Synchronous Sata Flow: Modular Code Generation from Hierarchical SDF Graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):83:1–83:26, Mar. 2013.

[26] M. Wetter. Co-simulation of Building Energy and Control Systems with the Building Controls Virtual Test Bed. *Journal of Building Performance Simulation*, 4(3):185–203, 2011.