A Method to Import an FMU to a Hardware Description Language

Min Zhang
Synopsys Inc., USA, minz@synopsys.com

Abstract

In this paper, a new method of importing FMUs (Functional Mock-up Unit) [1] to a multi-domain, mixed-mode simulator is presented. Supporting FMI (Functional Mock-up Interface) 2.0 for Model Exchange by converting an FMU to an HDL (Hardware Description Language) wrapper model not only takes advantage of the existing simulator capabilities, but also avoids a significant amount of work in the core of the simulator. The selected HDL in this paper is MAST which is used in both Saber and SaberHDL simulators [2][7]. To make the FMU import process easier, a general conversion utility, FMU2MAST, was developed which converts an FMU to a MAST model automatically. Two examples, bouncing ball and motor drive system are presented. With these two examples, three techniques used in this method are discussed: Accurate event detection in a variable time-step integration algorithm; Re-initialization of a state variable in MAST; and solving DAE (Differential Algebraic Equation) of a coupling FMUs system. This new FMU import method has been proved a success with 44 examples exported from five different tools.

Keywords: FMI, FMU, HDL, MAST, Modeling, Simulation, Saber, SaberHDL, DAE

1 Introduction

In order to improve the exchange of simulation models between suppliers and OEMs, FMI (Functional Mock-up Interface) is initiated by Daimler AG in 2010. It defines an interface to be implemented by an executable called FMU (Functional Mock-up Unit). It has been used in automotive and non-automotive industries, and supported by many simulation tools [1].

Saber is a multi-domain, mixed-mode simulator used in automotive and aerospace industries for more than thirty years. It supports models written by an HDL such as MAST and VHDL-AMS [4][7]. In recent years, there are more and more requests to import FMUs into Saber simulator from the industry.

To support the FMI 2.0 for Model Exchange in an HDL simulator, it requires a significant work in the core of the simulator. In this paper, a new method to import FMUs for Model Exchange is explored. Instead of supporting the FMU in a simulator core, the method proposed in this paper is to convert an FMU into a MAST wrapper model. This method has the following advantages:

1. Reduce significant work in the core of the simulator, which is time consuming as well as risky.
2. Avoid the duplicated work in another MAST simulator to support FMU import. Once an FMU is successfully converted to a MAST model, the generated MAST model works in Saber simulator, it also can work in another MAST simulator, SaberHDL, without any extra work.
3. The generated MAST wrapper model inherits all the features of the MAST language, and is applicable for all the existing simulator analyses, such as operating analysis, transient analysis, and advanced Monte Carlo analysis [7].
4. The generated MAST wrapper models can be used along with other models written in MAST or VHDL-AMS [4][6]. It increases the model availability and helps the study of a more complex and interesting system.

In chapter 2, a detail analysis of the data types and variables in FMU and MAST language is presented. Based on that, the equivalent objects in MAST language is derived. A new interface, saberFMI is introduced in chapter 3. It creates the communication channel between MAST models and FMUs through FMI interfaces. In chapter 4, A general conversion utility, FMU2MAST is introduced. It parses the model description file in the FMU, generates an equivalent MAST wrapper model automatically, which makes the conversion from FMUs to MAST models easier. Two FMU examples are presented in the chapter 5 and 6 to discuss the detail techniques used in the method. Although the bouncing ball example is simple, it is useful to discuss two issues: accurate event detection in variable-time step integration algorithm and re-initialization of a state variable in MAST model. The motor drive system is used to illustrate how a coupling system with multiple FMUs are solved in a DAE solver. The method has been verified in two simulators with 44 FMU examples exported by five different tools.

2 Model Types and Variables

Both FMUs and MAST models are designed to describe a mixed-mode, multi-physics system, consequently they share many interchangeable objects, which are discussed below [1][2].

2.1 Data Types

MAST has six data types. The four basic scalar data types are integer, number, string and enumeration. These types have the same definition as the data types defined in the FMUs. The difference is that the identifier name in the FMUs is case sensitive, and it also can have special characters, such as space, parenthesis, braces and brackets in it, e.g. "der(v)". In such cases, an underscore "_" is used to replace the illegal MAST characters, therefore, the variable name "der(v)" in the FMU will be translated into "der_v" in MAST. Due to lack of char and byte types in MAST, any FMUs with these variable types will not be supported for MAST conversion.

2.2 Model Connection Points

In MAST, the model connection points communicate the characteristics of the model with the rest of the system. There are three different types of connection points: continuous analog pin, event-driven port and data flow type.

The continuous analog data type presents the physical connection which has across and through units. For example, the electrical port has voltage as a cross unit and current as a through unit. The across and through units satisfy the KCL (Kirchhoff's Voltage Law) and KVL (Kirchhoff's Current Law) laws, thus the analog pin is energy conservative. There is no direction for this type. Currently FMI 2.0 doesn't support physical connection port. A method has been proposed to create an adaptor model for the physical port to solve the problem [8]. The proposal assumes the voltage as input, and current as the output, which may not applicable for all the cases. The better way is to solve the equations associated with the physical ports in a DAE solver, which may be supported in FMI/FMU standard in the future.

The event-driven port is used to communicate a model's discrete behavior in the system. It has three direction modes: input, output and inout (also called bi-direction). It is the input mode if it is driven by other discrete events and output mode if it drives other models. It is the inout mode if it adopts both behaviors. It is equivalent to the FMU scalarVariable with the variability of discrete and causality of input or output.

The data flow connection describes a model in a control flow fashion. It has input and output modes. It is input mode if it reads values from the connection point and output mode if it writes values to the connection. It is equivalent to the FMU scalarVariable with the variability of continuous and causality of input or output.

2.3 Model Parameter

Model parameters are coefficients that reside within physical characteristic equations which describe the model behavior. During simulation, the model parameters remain constant, however it can be varied in different simulation runs. The model parameter is equivalent to the FMU scalarVariable with the causality of parameter and variability of constant.

2.4 Constant Variable

The constant variable is similar to the model parameter, but used locally and only visible inside the model. Its value is constant, or may be calculated based on other model parameters but remains constant during the simulation. The constant variable in MAST model is equivalent to the FMU scalarVariable with the following types: 1. Causality is parameter and variability is fixed; 2. Causality is calculatedParameter and variability is fixed or constant; 3. Causality is local and variability is fixed or tunable.

2.5 State Variable

In MAST, the state variable (**state**) is used to describe the discrete behavior whose value remains constant between two consecutive time steps but may change from time point to time point. It is equivalent to the FMU scalarVariable with the following types: 1. Causality is independent; 2. Causality is local and variability is either discrete or tunable; 3. Causality is parameter and variability is tunable.

2.6 Local Analog Variable

A local analog variable in MAST (**val**) is a continuous variable and used to simplify the complicated system equations. It is equivalent to the FMU variable with the causality of local and variability of continuous.

2.7 System Analog Variable

In MAST, system variables (**var**) are the unknown variables that are needed to be simultaneously solved via the DAE (Differential Algebraic Equations) solver. Usually they are the analog connection points, data flow connections, through variable of independent source and **d_by_dt** operators for differential equations. FMU only solves the ODE (Ordinary Differential Equations), thus MAST system variable (**var**) is equivalent to an FMU continuous state variable with the causality of local and variability of continuous. The exchangeable objects between MAST and FMUs are shown in figure 1.

DOI
10.3384/ECP18154147

			FMI		
			causality	*variability*	*state*
MAST	*connection*	*input*	input	continuous	
		output	output	continuous	
		state (in)	input	discrete	
		(out)	output	discrete	
	variable	*parameter*	parameter	fixed	
		constant	calculatedParameter	fixed	
			local	fixed	
			local	constant	
		state	parameter	tunable	
			independent		
			local	discrete	
			local	tunable	
		val	local	continuous	
		var	local	continuous	yes

Figure 1 Exchangeable objects between the MAST and FMU

3 saberFMI Interface

In order to communicate with an FMU inside a MAST model, a new MAST foreign routine interface, saberFMI is developed. It exchanges the information between the MAST and FMU through the FMI. With this interface, the simulator can talk to FMUs through the interfaces: MAST, saberFMI and FMI. The communication interfaces between the FMUs and Saber simulator are shown in figure 2.

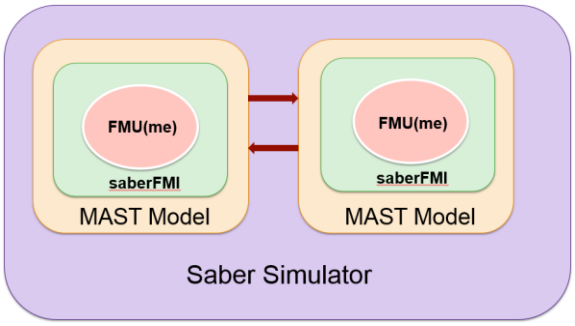

Figure 2 Interfaces between Simulator and FMUs

3.1 Parameter Section

The **Parameters** section is a group of sequential statements that initializes the system. It validates all the model parameters, calculates the internal constant variables which depend on the model parameters, and initializes state variables if needed. It is equivalent to **instantiated** and **initializationMode** states defined in FMI state machine [1]. Four saberFMI interfaces are introduced in this section: *initialization*, *setValues*, *updateValues* and *getValues*. The first *initialization* saberFMI interface calls **fmi2Instantiate** to construct the fmi2Component and return fmiHandle to the MAST model. The second *setValues* interface passes all the model parameters to the FMU by calling **fmi2SetX** internally (where **X** is one of the FMU data types, e.g. Integer, Real and Boolean). The third *updateValues* interface will update all the internal variables by calling the FMI interface **fmi2EnterIntializationMode** and **fmi2ExitInitializationMode**. The fourth saberFMI interface, *getValues*, will get all the internal local variables by calling **fmi2GetX** and pass them back to the MAST model. The saberFMI calling sequence in MAST model during the initialization phase is shown in figure 3.

```
Parameters {
  fmiHandle = saberfmi(initialization...) {
    fmi2Instantiate();
  }
  saberfmi(setValues,fmiHandle,...) {
    fmi2setReal/Integer/Boolean();
  }
  saberfmi(updateValues,fmiHandle...) {
    fmi2EnterInitializationMode();
    fmi2ExitInitializationMode();
  }
  h_ic = saberfmi(getValues,fmiHandle...) {
    fmi2GetReal/Integer/Boolean();
  }
}
```

Figure 3 saberFMI in MAST Parameter section

3.2 When Sections

The **when** section in MAST is used to construct the discrete state machine. Once the input events get changed, Saber simulator will call the statements in the body of the **when** section, and propagate the events until no new events are generated. This section is equivalent to the **EventMode** state in the FMI state machine. Two new saberFMI interfaces are introduced: *setEvents* and *checkEvents*. The first *setEvents* interface will call **fmi2EnterEventMode**, then **fmi2SetX** to update input

```
When(event_on(in1)) {
  saberfmi(setEvent,fmiHandle,...) {
    fmi2enterEventMode();
    fmi2setReal/Integer();
  }
  nextEvent=saberfmi(checkEvents,fmiHandle,...) {
    while(newDiscreteStatesNeeded) {
      fmi2newDiscreteStates();
    }
    fmi2enterContinuousTimeMode();
  }
  schedule_next_time(nextEvent)
}
```

Figure 4 saberFMI in MAST when section

discrete variables. Next, **fmi2NewDiscreteStates** will be repeatedly called to propagate all the events until no new event is generated. The FMI interface **fmi2EnterContinuousTimeMode** will be called later to appropriately switch back to **continuousTimeMode**. At the end of **when** section, the MAST built-in scheduling function, **schedule_next_time()**, will be called to force the next simulation time to be the value returned by *checkEvents*. The FMI calling sequence in MAST **when** section is shown in figure 4.

3.3 Values Section

In MAST, the **values** section of a model is used to transform variables into a form needed in the equations section. In this section, three new saberFMI interfaces are introduced: *timeValue*, *nonlinear* and *checkCross*. The first one, *timeValue*, passes continuous input values to the FMU by calling **fmi2SetTime** and **fmi2SetReal**. The second saberFMI interface, *nonlinear*, receives the nonlinear function values of the scalarVariables defined in the FMU by calling **fmi2GetReal**. Saber simulator will automatically construct the nonlinear functions, extract the numerical partial derivatives for each nonlinear dimension, and fill in system Jacobin matrix. The third saberFMI interface, *checkCross*, will get eventIndicator value, return zero if no cross event is detected, and one a cross event is detected. This evenIndicator will be used in a **when (threshold)** section to find the exact time when any event occurs and force the simulator to find a solution at that time point. All these steps are accomplished in the **continuousTimeMode** state. The FMI calling sequence in MAST **values** section are shown in figure 5.

```
Values {
  saberfmi(timeValue,fmiHandle,input...) {
    fmi2setReal ();
  }
  xIndicator=saberfmi(checkCrosss,fmiHandle,time...) {
    fmi2getEventIndicators();
  }
  der_h=saberfmi(nonlinear,fmiHandle,time,h,v) {
    fmi2setReal();
    fmi2completedIntegratorStep();
    fmi2getReal();
  }
}
```

Figure 5 saberFMI in MAST values section

4 FMU2MAST conversion

With the new saberFMI interface introduced in section 3, the FMU models can be represented by a MAST wrapper model and simulated in Saber simulator. It is nevertheless very challenging to translate an FMU to a MAST model manually because it requires advanced knowledge of both FMU and MAST. To assist the conversion process, a new utility FMU2MAST was created.

The conversion process can be divided into three steps. The first step is to read in the model description file, parse the XML file, and build up the XML tree in the memory. The second step is to preprocess the variables. First, all the illegal MAST variables will be renamed to valid MAST names; Next, the alias variables, which share the same **ValueReference** attribute, will be identified. The alias variables can be represented in MAST model with a simple assign statement without unnecessary FMI calls; At the end, the FMU2MAST utility will sort all the variables into the groups in the order of connection points, continuous state variables and others. The reason of this order is to avoid renaming the names of connection point and system variable in MAST, thus keep the most important variable names unchanged from the original names in the FMU. The third step is to convert the XML tree into the MAST data tree. The new MAST tree categorizes the variables into six categories based on data types shown in figure 1: **inputs**, **outputs**, **constants**, **states**, **vals** and **vars**. With all the equivalent information available, FMU2MAST can generate a correct MAST wrapper model.

This new FMU2MAST utility has been verified by 44 cross-check FMUs exported from five different tools: FMUSDK, OpenModelica, MathWorks, Dymola and standard reference tests suggested by [3]. The generated MAST models have been tested in two different simulators: Saber and SaberHDL. The simulation results match well with the reference results provided by the examples. The tests are selected with the intention of covering as many different applications as possible: 1. Exported from five different tools; 2. Analog system: vanDerpol; 3. Discrete system: BooleanNetwork1; 4. Mixed-Mode system: BouncingBall; 5. Multi-domain system: for example, hydraulic ControlledTanks, mechanical CoupledClutches, electrical Rectifier and thermal ControlledTemp; 6. Coupling system with multiple FMUs: motor drive example; 7. Complex

	vendor	input	output	discrete	state	others
Controlledtanks	OpenModelica	0	0	246	4	535
Motor_drive2	Dymola	0	1	39	15	253
MixtureGases	Dymola	0	2	4	16	76
DFFREG	Dymola	0	2	40	0	44
CoupledClutches	Dymola	1	4	50	18	123
Rectifier	Mworks	0	8	6	4	177
ControlledTemp	Mworks	0	2	4	1	64
BooleanNetwork	Mworks	1	9	57	0	29
FullRobot	Mworks	0	6	109	36	5259
BouncingBall	FMUSDK	0	0	1	2	3
vanderpol	FMUSDK	0	0	0	2	3

Figure 6 cross-check tests information

DOI
10.3384/ECP18154147

system: 36 state variables, 109 discrete variables and more than 5000 other variables in FullRobot example. The detail information of some tests are listed in figure 6.

5 Bouncing Ball Example

A simplified MAST model of a bouncing ball example generated by FMU2MAST is given in the appendix. This example has two state variables. Although it is very simple, it can be used to illustrate two important simulation techniques. The first one is the accurate event time detection. It is challenging to detect the accurate event time in a mixed-mode simulator with a variable time-step integration algorithm. The second is the re-initialization of a state variable. It is trivial if the HDL provides this capability, such as **reinit()** in Modelica [5] and **break** in VHDL-AMS [6]. However, MAST language has no such a function. If these two features are not implemented appropriately in MAST wrapper model, the bouncing ball will not behave correctly. Three bouncing ball examples exported from different tools have been tested, only two of them succeed. The reason for the failed one is that the state variable velocity has no <**reinit**> attribute in its modelDescription.xml. After the <**reinit**> attribute is added manually to the velocity variable, it works as well as other two.

In the header of the MAST model attached in the appendix, the height h is defined as an output connect point, the gravitational constant g is defined as a model parameter with default value of 9.81. The elastic coefficient e is defined as an internal state variable with initial value of 0.7.

At the beginning of **parameters** section, saberFMI *initialization* interface is called to create the fmiHandle for the FMU model. Inside this interface, it will call standard **fmi2Instantiate** to instantiate the FMU. The saberFMI *setValues* interface is called to pass the gravitational constant g to FMU and update it by **fmi2SetReal**. After that, saberFMI *updateValues* is called, the FMI calculated parameters will be updated by standard FMI function **fmi2EnterInitialization** and **fmi2ExitInitialization**. At the end of **parameters** section, saberFMI *getValues* is called to get the initial values, h_ic is initial value for continuous state height h and v_ic is for velocity v. They will be used in the MAST **control_section** to set initial condition for the differential equations in operating point analysis.

The first **when** section with sensitive variables **dc_init** or **time_init** is called at beginning of operating point analysis and transient analysis. It processes all the initial events by FMI interface, **fmi2NewDiscreteStates**, until no new events are generated.

In **values** section, two saberFMI *nonlinear* interfaces are called to get FMU derivative variables: der(v) and der(h). Obviously, the equation der(v) = -g has a constant relationship, and der(h) = v has a linear relationship. Both equations don't have a nonlinear relationship but a nonlinear MAST function is used here. The reason is that the FMU model is a black box to the MAST model, there is no explicit expression for each equation, however, the linear/nonlinear relationship is available in the **ModelStructure** section in the model description file. According to FMI documentation, if the **<dependencies>** attribute is presented as an empty list, the *Unknown* depends on none of the *Knowns*; If the **<dependencies>** attribute is presented and **dependenciesKind** is constant or fix, then the *Unknown* has a linear relationship with the *Knowns*; If no **<dependencies>** attribute is provided, then *Unknown* has no particular dependency on *Knowns*. In this example, since both der(v) and der(h) have no <dependencies> attributes, they are translated as nonlinear dependency of state variables: v and h.

In order to detect the accurate events time, an interface saberFMI *checkCross* is introduced in the **values** section. It checks whether there is an event occurred in the time interval between the last accepted time and current time. If it is true, then xIndicator becomes 1, and 0 if it is false. In this example, the true means the ball hits ground, the height variable h becomes negative and the velocity variable needs to be re-initialized as v=-e*v0. To achieve this, a MAST threshold detection section is used: **when (threshold (xIndicator, 0.5, before, after))**. This threshold function will use the simulator built-in threshold cross detection algorithm to find the exact time when the crossing event occurs. If the xIndicator crosses 0.5, a new event, *cross*, will be scheduled, which triggers another process **when (event_on (*cross*))** to force simulator to find a solution at this time point. With this build-in event detection method, the event time can be detected precisely with an

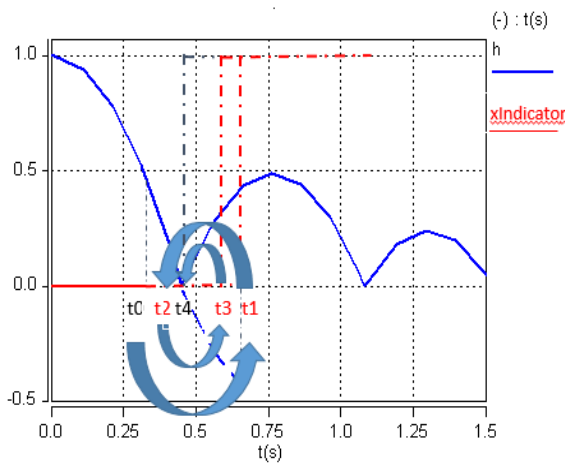

Figure 8 Threshold Crossing Detection

error less than one picosecond. The iterative event detect process can be shown in figure 8. Where t0 is the last accepted time point, t1 to t3 are the tentative time points during the iteration, although the xIndicator becomes 1, they are discarded because they don't meet the time criteria: $\Delta t \leq 1\ ps$. t4 is the final event time, it satisfies both event and time constraints: xIndicator=1 and $\Delta t \leq 1\ ps$.

After the eventIndicator is found to become true, the voltage v needs to be re-initialized due to the attribute **reinit=true**. It is trivial for an HDL if it provides the re-initiation capability, such as Modelica [5] which has reinit() function to do it. However, there is no such function in MAST language. To be able to simulate this dynamic re-initialization behavior for a continuous state variable, an additional equation (2) is introduced. For example, der(v) = -g, where g is a constant, the v decreases linearly with time, a single MAST differential equation d_by_dt(v) = -g cannot achieve this re-initialization behavior: v = -e*v0. To initialize it dynamically, a new variable v_0 is introduced to represent the original differential equation, another discrete variable v_{init} is used to help describe the re-initialization behavior. The original state variable v with **reinit** attribute now have two equations: (1) and (2). Solve these two equations together can achieve the dynamic re-initialization in MAST wrapper model:

$$\frac{dv_0}{dt} = -g \qquad (1)$$

$$v = v_{init} + \Delta v_0 \qquad (2)$$

The v_{init} is the value re-initialized immediately after each event is detected, Δv_0 is the velocity difference between the solution of equation (1) at current time t and time when cross event was detected. The final solution v should be v_{init} plus Δv_0. The solution for the state variable velocity with **reinit** attribute is shown in figure 9, the orange line represents the original differential equation (1) and the blue line represents the new equation (2) of a state variable with **reinit** attribute.

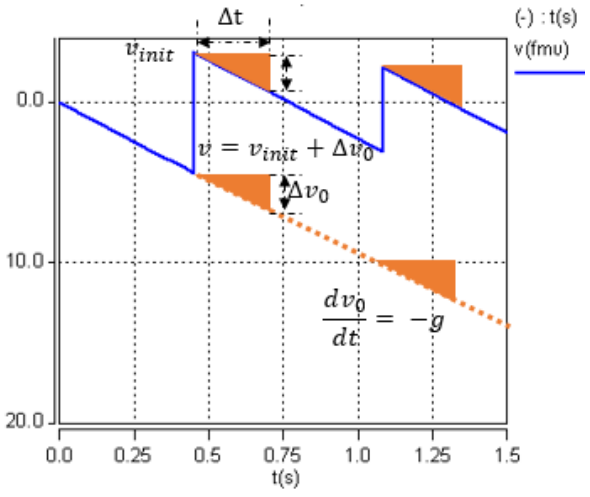

Figure 9 Solution for State Variable with reinit

All these detail works of the event detection and state variable re-initialization are handled in FMU2MAST utility during the FMU conversion. Figure 10 is the Saber simulation results of the MAST model converted from the FMU: ref_BRef.fmu [3]. The transient analysis uses variable time-step integration algorithm. The initial time step is 1 us, then gradually increase to 100 ms around 0.4 second when the height of ball is close to zero. After the height of the ball becomes negative, the eventIndicator of the FMU becomes 1, the converted MAST model will find this event, re-initialize the state variable v_{init} at this time and save the continuous solution v_0 of equation (1). With v_0 and v_{init} the actual velocity can be calculated with equation (2). After this event, the time step is reset back to the initial time step of 1 us, and gradually increases based on truncation error until the ball hit the ground again.

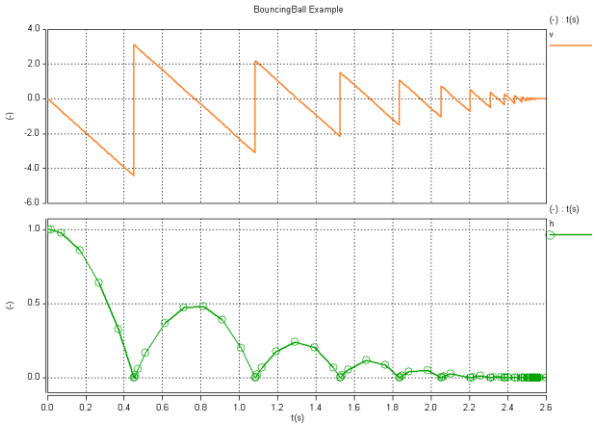

Figure 10 Bouncing Ball Results

Three bouncing ball examples have been tested. They are exported by FMUSDK, MathWork and [3]. Two of them worked well with this method but one failed. The reason for the failed one is that the velocity state variable doesn't have the attribute **<reinit>**, therefore, the translated MAST model doesn't know which variable needs to be re-initialized when the eventIndicator is detected. After manually adding attribute **<reinit=true>** to velocity in modelDescription.xml, the new generated MAST models works as well.

6 Motor Drive Example

When a system has multiple FMUs which are connected in a loop, the system needs to be solved by evaluating all the FMUs inside the loop repeatedly until the residue is close to zero [1]. This method works, but is difficult for a heterogeneous system that there are some non-

DOI
10.3384/ECP18154147

FMU models (MAST or VHDL-AMS) involved in the loop. To be able to handle more general applications, Saber simultaneously solves the DAE equations from all the models written in different languages. A typical DAE solver needs the partial derivatives of the equations to construct the Jacobian matrix during the iteration, and the FMI provides **fmi2GetDirectionalDerivative** interface for it. However, this interface is an optional in FMI standard, many FMUs don't provide it, except the FMUs exported by Dymola. It is known that one of the advantage of an HDL is that it does not require the modeler to provide the derivatives of the model equations. Saber simulator will analysis the MAST model, figure out the dependencies of all the linear/nonlinear equations, and approximate the nonlinear equations with PWL (Piece-Wise Linear) method [7]. With the PWL approximation, the partial derivatives can be obtained numerically without FMI interface **fmi2GetDirectionalDerivative**. This example will be used to discuss this method in Saber simulator.

This example is exported from Dymola. It has three FMUs: stimuli, controller and motor. The model connection diagram is shown in figure 11.

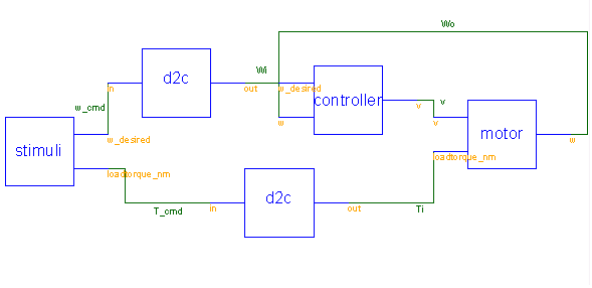

Figure 11 Motor Drive Schematic

The detail FMU implementations are embedded in binary code compiled from C source files. Even the FMU provides the original C files, it is still harder to understand the model characteristics when compared to an HDL. However, the model description file modelDescription.xml provides very helpful information, such as model inputs/outputs, state variables and dependency relationship of *Unknowns*. Based on this information, it is possible to derive the abstract equations of an FMU.

The first stimuli FMU sends out the reference angular speed and torque with respect to time. From its modelDescription.xml, the output equations can be derived as:

$$\begin{cases} \omega_i = u_1(t) & (3) \\ T_i = u_2(t) & (4) \end{cases}$$

Where ω_i is desired angular speed and T_i is the required torque.

The second controller FMU takes the desired angular speed ω_i from the stimuli, and the actual speed ω_o fed back from the motor as inputs, produces output voltage to the motor. From **<ModelStructure>** in the modelDescription.xml, it is known that the output angular speed depends on three variables with the same **dependenciesKind** of "fixed": the reference speed, the feedback speed and a state variable pi_x. The derivative $\frac{dpi_x}{dt}$ depends on both the reference speed and feedback speed with the same **dependeniesKind** of "fixed". According to the FMI documentation, the "fixed" **dependenciesKind** represents the *Unknown* depends on a *Known* with a fixed factor, and the factor is an expression that is evaluated before the **fmi2ExitInitializationMode** is called. Based on this, the characteristic equations of the controller model can be derived as:

$$\begin{cases} \frac{dpi_x}{dt} = k_1 * \omega_i + k_2 * \omega_o & (5) \\ v = k_3 * \omega_i + k_4 * \omega_o + k_5 * pi_x & (6) \end{cases}$$

Where pi_x is a state variable, v is output voltage for the motor, ω_i is reference angular speed from the stimuli, ω_o is the actual angular speed fed back from the motor, k_1 to k_5 are fixed coefficients. If $k_1 = -1$, $k_2 = 1$ *and* $k_3 = -1$, $k_4 = 1$, then the model is a classic PI (Proportional-Integral) controller, and the state variable pi_x is the integral of the speed error.

The third FMU, the motor model, takes the controller output voltage and stimuli torque command as inputs, delivers required torque and maintains the desired motor speed. From its **<ModelStructure>** in the modelDescription.xml, it is known that this FMU has one output variable and three continuous state variables, the characteristic equations of motor can be written as:

$$\begin{cases} \frac{di}{dt} = f_1(v, i, \omega) & (7) \\ \frac{d\omega}{dt} = k_6 * i + k_7 * T_i & (8) \\ \frac{d\theta}{dt} = \omega & (9) \\ \omega_o = k_8 * \omega & (10) \end{cases}$$

Where i is the motor current, v is the voltage applied on the motor, ω is the internal speed with unit of rad/ps, θ is the rotation angle which is the integral of angular speed, ω_o is the motor output speed, k_6 to k_8 are fixed coefficients. The motor current derivative $\frac{di}{dt}$ depends on the motor voltage v, current i and the motor angular

speed ω with the same **dependenciesKind** of "dependent". According to the FMI standard, if the **dependenciesKind** is "dependent", it means the *Unknown* depends on the *Known* without a particular structure. This "dependent" **dependenciesKind** is treated as a nonlinear dependency during conversion from an FMU to a MAST model.

There are two extra blocks, d2c, used in the design. They are hyper-models to convert the discrete output to continuous output. The reason for this is that the angular speed and torque outputs of stimuli are discrete outputs while the inputs of the controller and the motor are continuous inputs. It is illegal in MAST language to connect the different type of ports together. With these two hyper-models, it is possible to drive the continuous ports of the controller and the motor with discrete speed and torque output.

When all the three models are connected in a loop, there are 8 equations in total. Equations (3), (4), (6) and (10) are algebraic equations, while (5), (7), (8) and (9) are differential equations. When the design is loaded into Saber simulator, the simulator will setup these equations into the following DAE form [4][8]:

$$\boldsymbol{Ax} + \boldsymbol{E\dot{x}} = \boldsymbol{B(t)} \qquad (11)$$

Where

$$\boldsymbol{x} = \begin{pmatrix} \omega_i \\ T_i \\ pi \\ v \\ i \\ \omega \\ \theta \\ \omega_o \end{pmatrix}, \qquad \dot{\boldsymbol{x}} = \begin{pmatrix} \dot{\omega}_i \\ \dot{T}_i \\ \dot{pi} \\ \dot{v} \\ \dot{i} \\ \dot{\omega} \\ \dot{\theta} \\ \dot{\omega}_o \end{pmatrix}$$

the matrix $\boldsymbol{E}$ is:

$$\boldsymbol{E} = \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

It is a constant sparse matrix. It is also structural singular due to the diagonal value in the row 1, 2, 4 and 8 are zeros, then the equation (11) is a DAE system.

As stated earlier, equation (7) is a nonlinear equation and it needs to be linearized in an iterative form with Newton-Raphson method to be solved:

$$F'(X_{k-1})X_k = -F(X_{k-1}) + X_{k-1}F'(X_{k-1})$$

With the Newton-Raphson method, the Jacobin matrix $\boldsymbol{A}$ of equation (11) in the nonlinear iteration is:

$$\boldsymbol{A} = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -k_1 & 0 & 0 & 0 & 0 & 0 & 0 & -k_2 \\ -k_3 & 0 & -k_5 & 1 & 0 & 0 & 0 & -k_4 \\ 0 & 0 & -\frac{\partial f_1}{\partial v} & -\frac{\partial f_1}{\partial i} & 0 & -\frac{\partial f_1}{\partial \omega} & 0 & 0 \\ 0 & -k_7 & 0 & 0 & -k_6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -k_8 & 0 & 1 \end{vmatrix}$$

Also the iterative RHS (right hand side) ***B(t)*** of the equation (11) becomes:

$$\begin{pmatrix} u_1(t) \\ u_2(t) \\ 0 \\ 0 \\ -f_1(v_{k-1}, i_{k-1}, \omega_{k-1}) + \frac{\partial f_1}{\partial v} v_{k-1} + \frac{\partial f_1}{\partial i} i_{k-1} + \frac{\partial f_1}{\partial \omega} \omega_{k-1} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

All the partial derivatives in Jacobin matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ are unknown, and need to be calculated in the iteration. FMI standard provides **fmi2GetDirectionalDerivative** to get partial derivatives, but it is optional in FMI standard, and not available in all FMUs. A simple numerical differentiation method is used in Saber to calculate the derivatives.

$$f'(x) = \lim_{h \to 0} \left(\frac{f(x+h) - f(x)}{h} \right) \qquad (12)$$

In matrix $\boldsymbol{A}$, all the coefficients $k_1$to k_8 are fixed and won't change after **fmi2ExitInitalizationMode** is called. These constant coefficients are calculated in MAST parameter section when the design is loaded into simulator. The coefficients in 5th row of matrix $\boldsymbol{A}$ are the partial derivatives of the nonlinear equation (7), and varying during the nonlinear iteration. These coefficients are calculated in MAST values section with nonlinear PWL approximation. In the PWL approximation method, each dimension of a nonlinear function is divided into many small subdivisions, named **sample_points** [7], and in each subdivision, the nonlinear function is approximated with a linear function. The smaller the subdivision width h, the more accurate the nonlinear approximation is. The subdivision width h is controlled by **sample_points** specification, it can be adjusted by user when there is a need for better accuracy. All the MAST equations

DOI
10.3384/ECP18154147

translated from an FMU by FMU2MAST utility will maintain the same linear/nonlinear relationship in the FMU. With the PWL method, the DAE equations (11) can be solved simultaneously in Saber simulator.

Figure 12 are the transient analysis results of the motor drive example with a variable time-step integration method. The initial time step is 1 us. The reference angular speed Wi is set to 10 rpm at time 0.1 second. The actual motor speed reaches 10 rpm around 0.2 second, about 0.1 second delay from the reference. After another 0.1 second it settles down 10 rpm at 0.3 second. At 0.5 second, the stimuli model applies 3 Nms torque on the motor shaft, the voltage required to maintain the speed is reduced, as shown with the purple curve, the voltage drops from 6.5 volts to 6 volts after 0.6 second. In this example, the variable time-step integration method is used in transient simulation. As it is shown in figure 12, the marks on the purple curve v indicate the exact simulation time points during the simulation. Whenever a step change occurs on the input signals, the truncation error (LTE) of the differential equations increases. To control the accuracy of results, the simulator automatically reduces the time step to make the LTE less than simulator setting. The smallest time step is about 50 nanoseconds, 1/20 of the initial time step (1 us). After the motor reaches its steady-state speed, 10 rpm, the LTE becomes so small, the simulator automatically increases the time step to improve the simulation speed without sacrificing the accuracy. As it is shown, at 0.9 second, the time step has been increased up to 0.1 second, which is about 100000 times of the

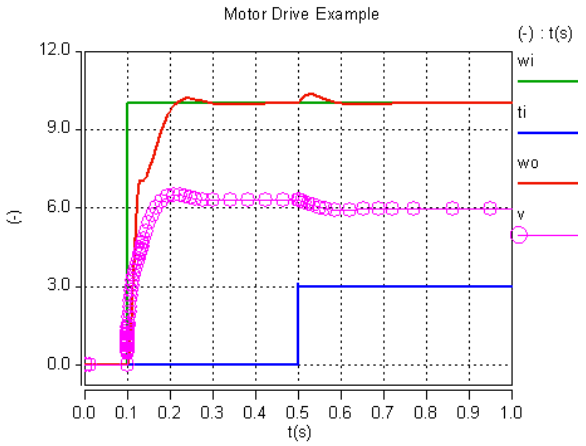

Figure 12 Motor Drive Results

initial time step (1 us). Compared to the fixed time step algorithm, the variable time-step integration algorithm significantly improves the simulation speed.

7 Conclusion and Future Work

In this paper, a new method of importing an FMU to a Hardware Description Language is introduced. A conversion utility, FMU2MAST, is developed to help the conversion from FMUs to MAST models automatically. This method has been proved a success with 44 FMUs exported from five different tools. With this method, the FMUs can be imported to another simulator which supports MAST language without any extra work. The converted MAST wrapper model can also be simulated with other non-FMU models which written in MAST or VHDL-AMS language to help the study of a more complex heterogeneous system. The FMI 2.0, the version used in the paper, doesn't support the general DAE system yet, however, this can be easily extended to support it when the FMI standard supports the DAE in the future. Right now this method is only applied to the FMI 2.0 for Model Exchange, but it can also be applied to support the FMU import for Co-Simulation as well in the future.

8 Appendix

Attached is the simplified MAST code generated from ref_bBRef.fmu. To make the model meaningful for illustration, only the relevant codes are kept.

References

1. Functional Mock-up Interface for Model Exchange and Co-Simulation, 2.0 July 25, 2014. https://www.fmi-standard.org/downloads
2. OpenMAST Language Reference Manual, 1.0, June 2004.
3. Christian Bertsch, Award Mukbil, Andreas Junghanns, Improve Interoperability of FMI-supporting Tools with Reference FMUs, pp. 533-540, Proceedings of the 12th International Modelica Conference, May 15-17, 2017, Prague, Czech Republic
4. R. Scott Coopper, The Designer's Guide to Analog & Mixed-Signal Modeling, March 1, 2001
5. Modelica – A Unified Object-Oriented Language for Physical System Modeling Language Specification, Version 3.0, September 5, 2007
6. Peter J. Ashenden, Gregory D. Peterson, Darrel A. Teegarden, The System Designer's Guide to VHDL-AMS: Analog, Mixed-Signal, and Mixed-Technology Modeling, September 10, 2002
7. Saber Simulator Guide: Reference Manual, Synopsys, June 2006
8. Yutaka Hirano, Satoshi Shimada, "Initiatives for acausal model connection using FMI in JSAE", Proceedings of the 11th International Modelica Conference September 21-23, 2015, Versailles, France
9. T. Blochwitz, M. Otter *et al.*, "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models", Proceedings of the 9th International Modelica Conference, September 3-5, 2012, Munich, Germany

```
# THIS MODEL IS NOT A COMPLETE MODEL. ONLY FOR DEMONSTRATION!
element template fmu_bbref h = g
number g=9.81 # acceleration of gravity.
{
	# variable declarations...
	parameters {
		fmiHandle = saberfmi(initialization,fmiHandle,instance(), "fmu_bBRef.fmu")
		constErr = saberfmi(setValues,fmiHandle,-1,g,g_id,fmiReal)
		constErr = saberfmi(updateValues,fmiHandle,g)
		h_ic = saberfmi(getValues,fmiHandle,h_id,fmiReal)
		v_ic = saberfmi(getValues,fmiHandle,v_id,fmiReal)
	}
	when(dc_init|time_init) {
		stateErr = saberfmi(initEvents,fmiHandle)
		v_init = v_ic
	}
	when(threshold(xIndicators,0.5,before,after) & after >0 ) {
		schedule_event(time,cross,1.0)
	}
	when(event_on(cross) & time_domain) {
		stateErr = saberfmi(updateCross,fmiHandle,time,h,h_id,v,v_id)
		nextEvent = saberfmi(checkEvents,fmiHandle,time)
		hasBreak = saberfmi(valuesChanged,fmiHandle)
		v_init = saberfmi(timeValues,fmiHandle,time,v_id,fmiReal)
		prev_v_0 = v_0
		schedule_next_time(time)
		e = saberfmi(getValues,fmiHandle,e_id,fmiReal)
	}
	when(dc_done|time_step_done) {
		stateErr = saberfmi(acceptValues,fmiHandle,h,h_id,fmiReal,v,v_id,fmiReal)
		e = saberfmi(getValues,fmiHandle,e_id,fmiReal)
		nextEvent = saberfmi(stepDone,fmiHandle,time)
		schedule_next_time(nextEvent)
	}
	values {
		der_h = saberfmi(nonlinear,fmiHandle,time,der_h_id,h,h_id,v,v_id)
		der_v = saberfmi(nonlinear,fmiHandle,time,der_v_id,h,h_id,v,v_id)
		xIndicators = saberfmi(checkCross,fmiHandle,time,h,h_id,v,v_id)
		delta_v_0 = v_0 - prev_v_0
	}
	control_section {
		initial_condition(h,h_ic)
		initial_condition(v_0,v_ic)
	}
	equations {
		h : d_by_dt(h) = der_h
		v_0 : d_by_dt(v_0) = der_v
		v : v = v_init + delta_v_0
	}
```


DOI
10.3384/ECP18154147