FUNCTIONAL MOCK-UP INTERFACE BASED SIMULATION OF CONTINUOUS-TIME SYSTEMS IN CADMIUM

Ritvik Joshi [a], James Nutaro [b], Bernard Zeigler [c], Gabriel Wainer [a] and Doohwan Kim [c]

[a] Department of Systems and Computer Engineering Carleton University
ritvikjoshi@cmail.carleton.ca, gwainer@sce.carleton.ca
[b] Computational Sciences & Engineering Division Oak Ridge National Laboratory Oak Ridge, TN, USA
nutarojj@ornl.gov
[c] RTSync Corp., Chandler AZ, USA
{zeigler, dhkim}@rtsync.com

ABSTRACT

The process for the standardization of the exchange of dynamic models resulted in the development of a Functional Mockup Interface (FMI). In this paper, we present a method to import a continuous time model using the FMI standard in Cadmium, a DEVS simulator. The method of implementation aims to address the challenges in importing and simulating the system developed in other tools into Cadmium. We implemented the simulation of the continuous-time model in DEVS by using a QSS solver. The paper also discusses the C++ library developed to support the implementation of FMU import. The DEVS implementation of the library and the solver were tested by the import and simulation of ordinary differential equation models.

Keywords: FMI, QSS, DEVS, Cadmium.

1 INTRODUCTION

Modeling and simulation play a vital role in the development of complex systems to address problems that include intricate interactions and dependencies between multiple elements. Modeling assists with understanding and conceptualization whereas simulation of these systems advantages us with validation and optimization. During the process of model development, the issue of sharing simulation models across various environments is commonly faced, especially if the system involves multiple domains. Differences in programming language and dissimilarity in the method of modeling are some of the causes of this issue.

Modeling tools are utilized by experts in various fields with limited exposure to programming and the theory of modeling and simulation. Navigation from one simulation tool to another tool to enhance the results is a challenging task. While researching how to solve this problem, the need for a new standard that could be used for exchanging models developed with different tools was identified. As a result, the Functional Mockup Interface (FMI) was defined as a standard for the exchange of dynamic models [3].

Besides the development of standards for exchanging models for collaboration, research efforts were also necessary to improve the overall performance of the simulation of such models. The enhancement of the performance results of the simulation can be achieved by improvement in accuracy and simulation time. Simulation time is dependent on the required number of calculations and the computation power of the device. The number of calculations for the simulation of a continuous-time system depends on the number of instances created during the process of discretization. Recently, a new set of methods showed potential as a method of discretization based on events. These methods generate fewer computational instances than the method of discretization of time [7], and they allow seamless integration with other formal models, including, in particular, Discrete Event System Specifications (DEVS) [2].

The DEVS formalism, introduced in [2] allows the specification of discrete event models in a simple and modular way. Simulation of a continuous time system on DEVS using a Quantized State System (QSS) [1] gives advantages such as reduction in computations and less simulation time.

Models developed in DEVS-based simulators like Cadmium [5] can be easily simulated using the same tool but as we discussed earlier, there is a need to improve the sharing of simulation models. This research shows a method that will allow models developed in other tools supporting FMI to be imported and simulated in Cadmium. To do so, we have implemented FMI import functionality in Cadmium. The idea is to import continuous time models using Open-Modelica [16], a simulation tool that supports FMI export functionality, into Cadmium. To do so, we developed a library in Cadmium that will allow importing models in a Functional Mockup Unit (FMU) [3]. FMU is an entity through which the FMI standard is implemented. The simulation of a continuous-time system in DEVS was performed by developing a QSS solver [1] in Cadmium. We present case studies to show the import and simulation processes, which currently include the continuous time equations into the model which can be further expanded to the import of hybrid models by the addition of an event detection mechanism into the simulator. The method of implementation along with a background is discussed. The process outlined can provide guidance for the development of FMI import on C++ or other object-oriented programming languages, within DEVS-based simulation frameworks.

2 RELATED WORK

The study of physical systems is a very important part of scientific and engineering problems, and we need tools to conduct such studies. Within the domain of engineering challenges, physical systems play a pivotal role as they serve as the foundation for complex systems. These systems have a diverse range, incorporating mechanical structures chemical processes, electrical circuits, and several other disciplines. Understanding the complex behavior of these systems has usually relied on the utilization of continuous-time equations [14]. These equations provide a mathematical description that coherently describes the evolution of system states over time. The resolution of such equations requires computational solutions, and these simulations provide insights related to the response of systems under diverse conditions and potential concerns.

Continuous time equations are commonly characterized using continuous system modeling methods, like bond graphs [12], and differential or algebraic equations. Differential equations are fundamental mathematical tools for representing continuous-time equations. Ordinary differential equations (ODEs) are equations with a single independent variable and differential algebraic equations are a combination of ODEs and algebraic equations [13].

The use of analytical methods for solving continuous system problems faces challenges or makes it impractical in certain types of problems. In specific cases, like with differential equations, solving complex systems with non-linear or time-varying dynamics is extremely difficult or impossible. Analytical methods commonly assume a deterministic behavior and steady-state and do not consider the variability in the real world. Simulation methods, instead, can handle complex dynamics, non-linearity, and discontinuity.[26][27]

Various classical methods have been defined to simulate continuous-time equations [9]. Discretization of the signals was inevitable for the implementation of continuous systems to achieve compatibility with digital technology unless analog devices were used. Analog devices had uncertainty issues because of physical drifts, external noise, etc. This led to the development of methods capable of handling continuous signals for simulation. Though the systems were continuous in nature, these methods used a discrete-time description for the basis of operation. The foundation of these methods is linked to sampling theory [10] which deals with converting continuous time signals into discrete time signals. The discretization of time offers conveniences like limiting the differential and integral part in computation and the feasibility of implementation on digital computing systems.

Numerical methods (such as Euler, Runge-Kutta, etc.) [9] were developed that could solve differential equations, especially when closed-form solutions are very difficult. These methods were applied for the analysis and resolution of problems in various fields such as physics and engineering, chemical engineering, etc. These techniques were based on time discretization. The methods were using iteration rules and symbolic manipulation. The methods were giving a solution obtained from a differential equation system defined in some discrete instances. Later, different efforts were taken to optimize the classical methods to minimize the computational requirements while meeting the accuracy specified by the user [11]. The constant time sampling was redundant for some cases where variable step versions were developed.

The simulation of continuous-time equations in numerical methods is based on an approximation procedure. The accuracy of the models is directly proportional to the number of samples in unit time which results in a higher number of calculations for high accuracy. The simulation time is also subject to the number of computations and available hardware. Obtaining optimum simulation results is a tradeoff between simulation result accuracy and time. Time discretization has limitations in dealing with input changes that occur between the two discrete instances as it has no access to the inputs between the two instances. Control over the simulation is lost between the subsequent instances. The chances of misreading the sequence of events are higher if multiple events occur between consecutive instances.

Despite the limitations in time discretization, the need for simulation of continuous systems on a digital device enforces discretization. A digital device permits a finite number of instances to be computed within a finite time interval. Taking the limitations of time discretization and the need for digital simulation into account, DEVS (Discrete Event System Specification) formalism [2] was developed in the mid-seventies. One of the advantages of DEVS is the reduction of the number of computational instances, which improves simulation time as it targets only the event instances for simulation instead of time sampling the signal. It also provides an abstract mechanism for event detection which helps deal with the event at the exact time instance, ensuring the accuracy of results and providing a more reliable representation of system dynamics.

A system can be described in DEVS if its input/output changes can be described by a sequence of events. An event is an instantaneous change in the system. It is defined by a value and time of occurrence. A continuous system can be modeled using DEVS with atomic and coupled models. An atomic model describes the behavior of the system by defining the inputs, outputs, transition of states, and its response to external events. The description of complex systems using only atomic models is difficult, but complex systems can be described as a combination of multiple simple systems. Such models are coupled models. The coupled model describes the behavior of a system using a composition of atomic and coupled models. The coupling helps to transfer the output events from one model to the input events of another model. Internal couplings have a connection between inputs and output ports corresponding to different models whereas external couplings are used to connect the ports of network to models.

DEVS became more popular among the community for the simulation of large systems involving computer systems and communication because of its advantages over discrete time methods. Distributed simulation became a popular form of simulation during the late 1990s. This resulted in demanding a tradeoff between accuracy, speed of computation, and communication resources. Techniques like dead reckoning were established but the studies were observational in nature which motivated Zeigler and Lee [7] to develop a theoretically rigorous technique that converts continuous quantities into discrete packets. The research presented an approach to represent DESS (Differential Equation System Specification) in DEVS. A significant change in input values was defined as an event using the concept of a quantizer, an event detector based on input and logical conditions to decide a significant change in inputs, outputs, or state variables.
For the representation of continuous time equations in DEVS, time discretization was replaced by state quantization which was termed as quantization of state variable. It is performed using a quantization function in which the concept of quantizer is implemented [1]. The concept of quantization of continuous time equations influenced the method to conduct numerical simulations of differential equations. Kofman,

Lee, and Zeigler [6] explain how the differential equations can be represented using DEVS. The quantization method not only reduces the number of calculations but also makes it easier to implement parallelization [15].

The approach of simulating continuous systems based on DEVS was further studied by Kofman and Junco [1]. They introduced Quantized State Systems (QSS), a new class of dynamic systems. QSS was defined as a continuous systems model where the quantization method converts the trajectory of state variables into piecewise constant functions. The quantizer function was also used to generate a piecewise linear state trajectory based on piecewise constant inputs. The authors also include the concept of hysteresis in the simulation, which solved the problem of the infinite number of transitions and legitimacy. QSS can be represented as a DEVS model and can be simulated using a DEVS-based simulator, or it can be simulated by a standalone QSS simulator [25]. State events are used to represent the partition block boundaries.

Modeling tools provide an interface for model development to the field experts, simplifying the development of modeling and simulation studies. Several general-purpose simulation tools were developed as well as domain-specific tools in the past few decades. The issue with multiple tools was that the model implemented in one tool was unable to be used in another tool [18].

Understanding the importance of standardization of modeling languages, the development of modeling language Modelica was initiated in 1997. Modelica [4] is a robust open standard modeling language used for object-oriented modeling. Modelica allows you to model using differential equations, algebraic equations, and discrete equations. It is a single language that can be used for building models across various domains like electrical circuits, hydraulics, mechanics, etc. The semantics is specified to translate the model using a set of rules into a hybrid differential algebraic model. A large set of freely available libraries are present in Modelica. Modelica-supported simulation environments like Dymola, and Open Modelica [16] are present and used for modeling, compiling, and simulation of Modelica models.

Even though the development of Modelica was aimed at exchanging models, the development of language support for Modelica-based tools was complex. Another challenge related to the exchange of models was protecting the product information which was retrievable from the physical model [3]. This created the need for a tool-agnostic standard for model exchange. Consequently, a standard model interface called FMI (Functional Mockup Interface) was published in 2011 for the exchange of models and for co-simulation. This was developed for standardizing data exchange across multiple dynamic models which are developed using different simulation environments [3]. The standard was developed considering model exchange and co-simulation. Using FMI for model exchange focuses on packing the dynamic models defined using discrete, differential, and algebraic equations in a format that can be shared and utilized by other environments or tools. In the case of co-simulation, two or more simulation tools are coupled together to form a co-simulation environment.

A Functional Mockup Unit (FMU) is a component used for implementation using the FMI standard. FMU is a container for a model exported by the modeling environment and is compressed into a single zip file. FMU consists of an XML file that contains the definitions of all variables inside FMU exposed to the environment, C-functions for the model equations, and object libraries or DLLs. The improved version of the standard was published in [17], which combined the model exchange and co-simulation interfaces into one standard.

Integration of FMI with a DEVS simulator has been investigated by multiple researchers in recent times. Müller and Widl [19] presented a methodology for linking FMI with discrete event systems for the simulation of cyber-physical systems while achieving modularity and heterogeneity. The formal way to interlink the discrepancy between the semantics of heterogeneous modeling formalisms and FMI was examined by Tripakis [22]. Experimentation to import a model through FMI and its simulation using a standalone QSS solver was shown in [23] and [24]. In this paper, we imported a continuous time model FMU as an atomic model and then simulated it using the QSS solver implemented in DEVS.

FMI was used to transfer a model developed in the Modelica-based modeling tool to a DEVS-based simulation tool. A wrapper was developed to extract the variables and functions from the FMU. In this research, we implemented an FMI import library to implement a wrapper for importing an FMU into an atomic model which was further simulated using a DEVS-based simulator. We developed a model using Modelica language and exported the model as an FMU. This FMU was transferred to a DEVS-based simulator where the FMI wrapper imported the FMU and simulated it by using the QSS solver. For the development of models in Modelica, we used Open-Modelica [16], an open-source Modelica and FMI-based modeling and simulation environment. The extended version of Modelica was used for the development of this environment. The tool is also capable of developing Modelica code from graphical models. The tool assists in implementing and validating the FMUs. A C++-based wrapper was developed in Cadmium [33], a DEVS simulator is a header-only C++17 library that is easy to integrate into different projects. It provides a robust application programming interface for using the available features for modeling and simulation in DEVS. Atomic and coupled models need to be defined at the user level to define a DEVS model. The atomic model is declared as a class and input output ports of the atomic model are declared as a structure. The coupled model is also defined using class. At the user level, Cadmium provides a set of methods such as *internalTrasition, externalTransition, output,* and *confluentTransition* for the definition of atomic models and *addComponent, addCoupling* for the definition of coupled models. The abstract simulator defined in [28] was used for the implementation of the simulation algorithm in Cadmium. The task of simulation is performed with the help of two components, simulators and coordinators. The root coordinator is responsible for initializing and advancing the simulation. The instance of the model is shared with the root coordinator for simulation. Cadmium also has various advanced features implemented for model verification and reducing the possibility of errors. Cadmium maintains the modular architecture of the simulator which supports the natural structure of a DEVS model. The values of state variables after every simulation step can be recorded in the simulation log obtained in the form of a CSV file. The connection between the root coordinator and the clock in the tool allows it to simulate the model in real-time. Cadmium supports multiple data types for defining Time and Messages. As the library is compliant with C++ 17, it can be compiled on multiple platforms like Windows and Linux. As FMI inherently supports C++ the transfer to a C++-based library was natural.

3 FMI IMPORT AND DEVS SIMULATION METHOD

This section discusses how to import FMI and how to simulate the model using the Cadmium tool. The definition of the software stack includes three parts. First, the definition of an FMI library to import FMUs into the DEVS Cadmium toolkit. Then, a QSS solver is used to represent continuous models and execute simulations in DEVS. Finally, the simulation of the model was imported in Cadmium. Each subsection gives a detailed description of each of these parts.

3.1 FMI importer

In this section, we will discuss the method of FMI importer implementation in Cadmium. The FMI importer is a software component that allows the integration of an external simulation model in an FMU into a simulation environment. The FMI standard comes with a set of robust header files that support the development of import-export functionality in the standard. By referring to the calling sequence state machine in the standard document, we implemented the model import functionality in the atomic model by a sequence of steps – instantiation, configuration, model execution, and termination.

An instance of FMU is created inside an atomic model for model import, which is supplied with model information like model name, number of state variables, and event indicators provided in modelDescription.xml along with the path to dynamic libraries inside the FMU. An atomic model inherits methods defined for the interaction with FMU.

The configuration step is performed immediately after the instantiation where FMU gets initialized by entering the initialization mode which sets all attributes and then enters checks for the events defined in

FMU. Inputs and output ports are defined in the importing atomic model based on the variable description in modelDescription.xml which gets updated during the model execution.

The execution of a model starts after the configuration step where the FMU instance enters continuous time mode. This step is implemented inside the internal transition function of the importing atomic model. The FMU time is updated with the current simulation time. The updated values of variables are fetched from the FMU and are used to set values of state variables inside the atomic. The rate of update depends on the time advance defined inside the atomic model.

The FMU might not be required for the complete simulation time. The instance of FMU might be an overload for the resource-constrained systems. This instance can be released by the atomic model and instantiated whenever required.

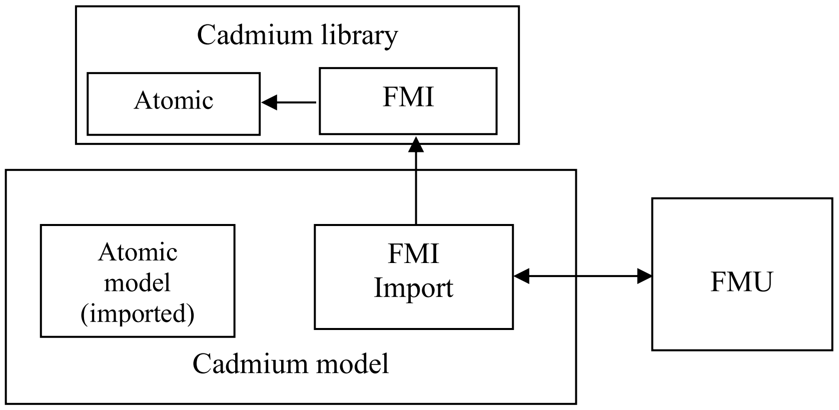

Figure 1(a): Block diagram of FMI importer.

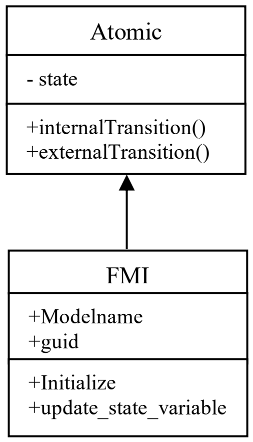

Figure 1(b): UML diagram showing hierarchy.

Figure 1(a) shows the block diagram of the FMI importer. The FMI block is a class developed in Cadmium. FMI Import is the class inherited from FMI which is included in the model developed for importing the FMU. FMU is the external model that is accessed by the FMI importer. The Atomic model is the block where the model is getting imported. This atomic model block will be further simulated. Figure 1(b) shows the UML class diagram for the FMI and Atomic class which is present inside the Cadmium library. The FMI class is inherited from the atomic class.

The DEVS formalism uses a modeling hierarchy to systematically decompose complex systems. Atomic models are at the lowest level of hierarchy which are then followed by coupled models. For the implementation of the steps discussed above, an FMI class was added to the modeling hierarchy where the methods for importing FMU are defined. FMI class inherits the atomic class and gets access to the public methods like internalTransition(), and externalTransition() for implementing FMU in DEVS. The FMU class defines the public variables used for model description which includes *modelname* and *guid,* which are used for model identification. The public variable *num_state_variables* is used to define the number of state variables declared in the model. The number of event indicators gives the number of events to be detected during the simulation time defined by FMU. The variable *so_file_name* stores the path to the shared library file in the FMU. This is seen in the following code.

```
template <typename S> class FMI: public Atomic<S> {
  public:
   FMI(const std::string& id, S initialState, const char* modelname, const char* guid,
            int num_state_variables, int num_event_indicators,
            const char* shared_lib_name, const double tolerance = 1E-06, //1E-8
            int num_extra_event_indicators = 0, double start_time = 0.0
            );
        virtual void internalTransition(S& s) const = 0;
        virtual void externalTransition(S& s, double e) const = 0;
        virtual void output(const S& s) const = 0;
        virtual double timeAdvance(const S& s) const = 0;
        virtual void initialize(S& s);
        virtual void update_state_variables(S& s) const; //S& s
        virtual void iterate_event(S& s) const;
        double get_time() const { return current_time; }
        double get_real(int k) const;
        void set_real(int k, double val);
        int get_int(int k);
        void set_int(int k, int val);
        bool get_bool(int k);
        void set_bool(int k, bool val);
}
```

The FMI class implements the functionality needed to instantiate FMUs. The methods required for the model execution (explained earlier in the section) are implemented inside the class. The method *initialize()* completes the configuration step where it sets up variables with initial values, sets the simulation time, and checks for the events before the start of simulation. The methods *get_real()* and *set_real()* are used for exchanging variables representing real numbers. Similarly, *get_int()* and *set_int()* work for integers, and finally *get_bool()* and *set_bool()* work for Booleans. The method *update_state_variable()* is for modifying the state variables after an event is detected. The *iterate_event()* method updates the *event_info* structure to update the parameters for events and checks the need for the change of discrete state based on the *newDiscreteStateNeeded* flag from the *event_info* structure. The *internalTransition()* updates the time for FMU based on the current simulation time and then reads the updated values of state variables. The *externalTransition()* method is called after the detection of an event where the continuous state of the system is updated.

The implementation of FMU importing functionality inside a DEVS model was achieved by defining the FMI_IMPORT class. This class communicates with the FMU and shares the variable values with the atomic model. The atomic model performs the numerical integration of the variables. The atomic model accesses the methods defined inside FMI through the FMI_ IMPORT class. E.g. `FMI_IMPORT::get_real(1);` for reading the value of state variable no 1.

For standardization of the function calls for the handling of FMU we developed a C++-based FMI library in Cadmium that supports model exchange specifications. The library is designed to support FMI specification 2.0. The library provides an implementation of generic functions used for the import of FMUs in the atomic model. The library uses the C++ template class feature for writing generic classes. The library uses dynamic pointers to handle the shared libraries for calling the methods required for FMI import. The library supports the exchange of variables of multiple datatypes which are stated in the standard. The library is developed on top of the atomic model library which permits seamless implementation of the import functionality explained above.

3.2 QSS solver implementation

Importing FMU inside the modeling environment gives access to the changes in variables according to the equations defined in the imported models. As we have seen before, QSS was developed for the simulation of continuous-time equations in DEVS. We implemented a first-order QSS solver [1] for the numerical integration of the quantized variables. After importing the values of state variables inside the internal transition function, it was passed to the first-order QSS solver. The QSS solver implementation in Cadmium was deciding the quantized state and it was also generating the value of time advanced based on the quantum size and the value of the derivative of the state variable. Hysteresis was also included in the implementation after the issue of infinite events was observed. The QSS solver is implemented using methods QSS1_with_HYS() which has access to the state variables. It reads the change in the value of the state variable, computes the integrated value and time advance, and updates the respective variables.

A constant quantum size was defined for the discretization of state variables using uniform quantization. As the derivative of the state variable was accessible through FMI, it was used to compute the time advance for the desired quantum size. Hysteresis was also included in the implementation after the issue of infinite events was observed.

The QSS solver was implemented using methods QSS1_with_HYS() which had access to the *state* variable which was used to define the structure of state variables. The *x_value* variable was the state variable that was being quantized. The slope was defined using *test_derivative_val* and the previous value of the state variable was used to define the value of *sigma* (time advance). The value of *x_value* was updated based on its previous value, *sigma,* and the derivative value.

The code snippet shown below shows a part of the implementation of the QSS solver in an atomic model.

```
void QSS1_with_HYS(Quantise_input_State& state) const        {
    static double last_p_der_val=0,last_n_der_val=0;
    state.x_value = state.x_value + state.sigma*state.test_derivative_val;
    if ((state.test_derivative_val > 0) && ((last_n_der_val==0)
                  ||(last_n_der_val>=QUANTUM_SIZE+HYSTERESIS)))  {
                state.sigma = (QUANTUM_SIZE)/state.test_derivative_val;
                last_p_der_val = state.x_value;
    }
    else if ((state.test_derivative_val < 0) && ((last_p_der_val==0) ||
             (last_p_der_val>=QUANTUM_SIZE+HYSTERESIS)))    {
                state.sigma = (QUANTUM_SIZE)/abs(state.test_derivative_val);
                last_n_der_val =  state.x_value;
            }
            else {
                std::cout << " SIGMA INFINITE " << std::endl;
                state.sigma = std::numeric_limits<double>::infinity();
            }
         }
```

3.3 Simulation of imported model in Cadmium

As we saw in the first section, a class FMI import is created inside the model. The exported FMU is accessed by the FMI importer for extraction of model information as we discussed before. The importer exchanges variables with atomic models based on the requests by the atomic model through *get_value* or *set_value* methods. The *get_value* method provides the values of the shared variable which can be used to obtain the present value of the state variable and its derivative. This derivative is then used for the prediction of the time when the state variable will be increased to the next quantized state. At this point, the model performs integration of state variables and requests new values from FMU by providing the value of time. This way it performs time synchronization with an FMU while performing discrete event simulation. The atomic model is then connected to other atomic models required for the simulation of the complete system through input and output ports to form a coupled model. This coupled model undergoes simulation with the assistance of the provided clock when it is shared with the root coordinator [33]. If there is an external event to the atomic model, then it calculates the value of the state variable by approximation using the previous state.

4 ODE DEFINITION IN OPEN MODELICA AND SIMULATION ON CADMIUM

In this section, we show an example of how the tool stack can implement the functionality of importing continuous time models using FMI, and its execution in Cadmium. The case study shows how to import an ordinary differential equation (ODE) model and simulate the model using a DEVS tool like Cadmium. We defined an ODE using Open-Modelica and exported an FMU, which was simulated using the simulator of Open-Modelica to generate the benchmark results for reference. The model was then exported as FMU 2.0. The following code snippet shows a section of the Modelica code for representing input, output, and a differential equation.

```
Model test
    Real a(start = 1.0) "state variable"
    Modelica.Blocks.Interfaces.RealInput u "input variable";
    Modelica.Blocks.Interfaces.RealInput y "output variable";

equation
    der(a) = u-a;
    y = -a;
end test
```

A new project was automatically created in Cadmium using a reference template to import the FMU. A coupled model was created, which includes a Generator atomic model and another atomic model to quantize signals called *quantise_input*. The generator model was created to send input *u* to the *quantise_input* model whereas the *quantise_input* model was importing FMI and integrating with the QSS solver. Instances of imported models were created in the *quantise_input* model using a shared library. The input and output ports were configured for the *quantise_input* model and the input was connected to the output of the generator. The internal transition function was used to read the values of all four variables using the *get_real* method. The received values were passed to the QSS solver function which was used to define the quantized state for the state variable. The quantum size was 0.1. hysteresis was also set to 0.1. The QSS solver was integrating the value for the state variable. The values of simulation time, time advance, state variable, output, and derivative of state variable were logged using the logger in Cadmium. This logger stores the value in CSV format which is easier to analyze. The simulation results were then plotted and compared to the benchmark results observed on the visualization window of Open-Modelica.

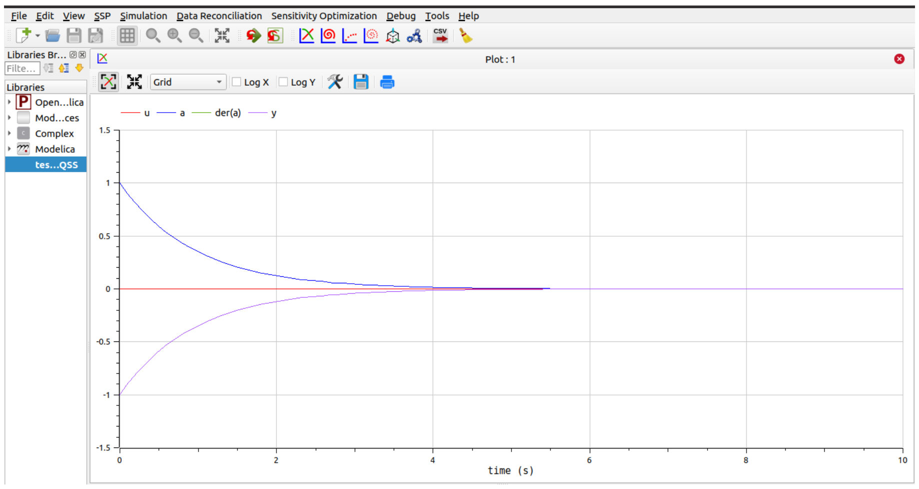

Figure 2: Visualization results in Open-Modelica with input u, state variable a, and output y.

Figure 2 shows a graph displaying input, state variables, and output. The topmost curve that starts at 1 shows the values of "*a*" which is the state variable of the model. The flat line at 0 shows the constant value of "*u*" which is the input of the model and the curve at the bottom that starts at -1 shows the value "*y*" which is the output of the model.

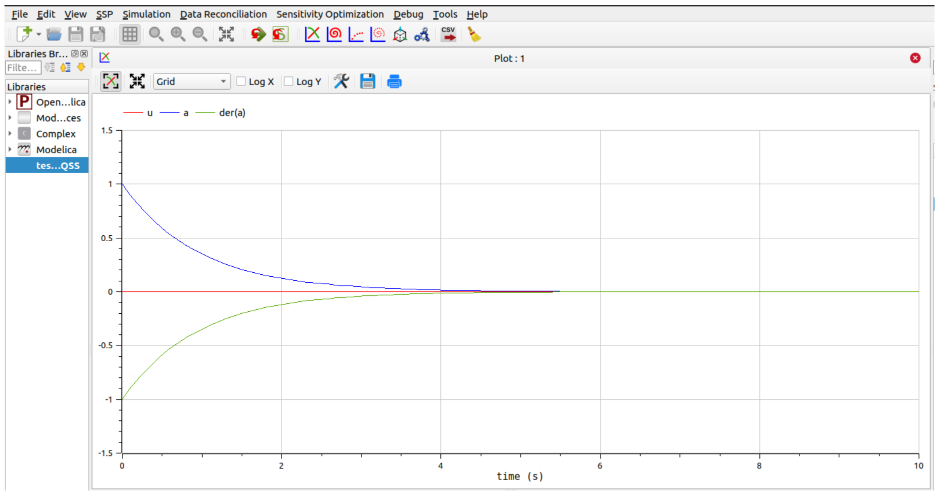

Figure 3: Visualization results in Open-Modelica with input *u*, state variable *a*, and derivative of state variable *der(a)*.

In Figure 3 the values of input "*u*" and state variable "*a*" are the same as it was shown in Figure 2. The curve at the bottom which starts at -1 shows the values of "*der(a)*" which is the derivative of the state variable. Both the graphs (Figures 2 and 3) are the result of the same test but show different parameters.

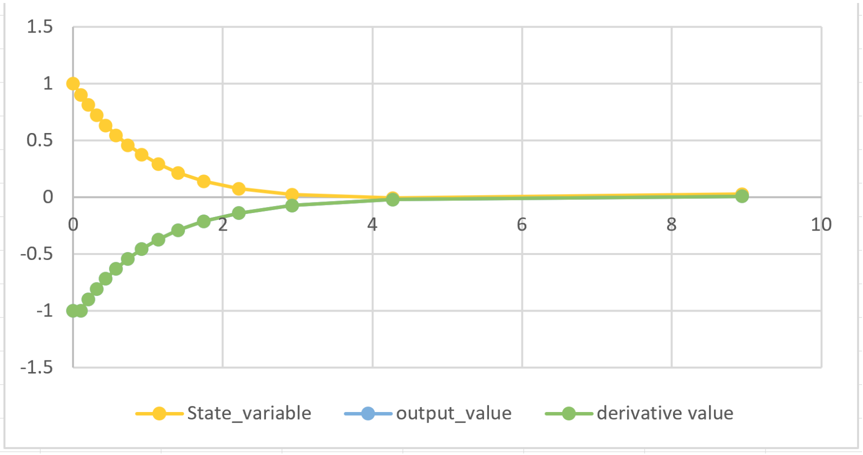

Figure 4: Visualization of Cadmium log generated in Microsoft Excel.

Figure 4 shows the graph of state variables at the top and the graph of derivative state variables at the bottom. The graph is a plot using the values obtained in the log file of the simulation.

As we can see in Figure 2, "*a*" is the state variable which is initialized to value 1. The value of "*u*" received at the input is kept at 0. We can observe the descending curve of the state variable "*a*" with time. As the input is 0 during the simulation time it shows a flat line, but that makes the value of the derivative of "*a*" the same as "*a*" but with the opposite sign which we observe in Figure 3. The value of output "*y*" is the same as the values of the derivative of "*a*". When observing this in the log generated by Cadmium, the values of state variable "*x*" showed a descending curve that reached a value of 0.007 which is close to zero at 4.2 sec. The curve observed in Figure 4 reads the values only at the quantized state with a quantum size of 0.1. The dots in the curve are the points where the quantization state was updated. As the curve between the two points is approximated based on the last value and the derivative calculated, the accuracy of this curve is dependent on the quantum size selected. The visualization results obtained in Open-Modelica closely resemble the visualization results generated in the last figure using the Cadmium log. We can also observe that the value of the derivative of the state variable is updated at the same instance where the value of the state variable is updated which is the time instance at which the internal transition function was executed.

We now discuss the import and simulation functionality of a different case study, where the differential equations used for the representation of the RC circuit are used for the simulation with AC input source and DC input source. The circuit diagram in Figure 5 was considered for simulation where a resistor with value R and capacitor with value C are connected in series with a DC supply voltage with value V. Current I is flowing through the circuit.

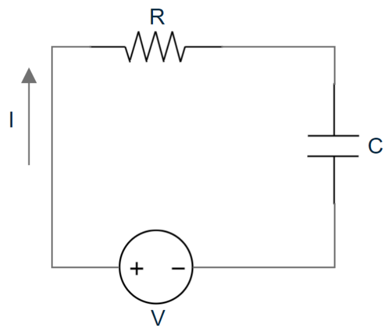

Figure 5: RC circuit diagram.

The modelica model that represents the value of voltage across the resistor (V_R) and across the capacitor (V_C) is shown in the following code snippet.

```
model SeriesRC
  parameter Real R = 20; // Resistance (ohms)
  parameter Real C = 0.01; // Capacitance (farads)
  Real V(start = 0); // Supply voltage (volts) 10
  Real I(start = 0); // Current through the circuit
  Real V_R(start = 0); // Voltage across the resistor
  Real V_C(start = 0); // Voltage across the capacitor
equation
  //input voltage
  V = 10;
   // Kirchhoff's voltage law
  V = V_R + V_C;
  // Ohm's law for resistor
  V_R = R * I;
  // Capacitor voltage equation
  der(V_C) = I/ C;
end SeriesRC;
```

The Modelica code for the DC input was tested using Open-Modelica. The value of resistor R was initialized to 20 ohms and the value of capacitor C was initialized to 0.01 farads. Later for the first experimentation we set the value of Voltage to 10V DC and simulated the model in Open-Modelica. A Model Exchange FMU was generated and then a new template project was created in Cadmium, and this FMU was imported to the project (using FMI_IMPORT method explained earlier), and simulated with a quantum of 1. Figure 6 shows the values observed during the simulation. The graph shows the values of current, voltage across the resistor, voltage across the capacitor, and its derivative. The plot of voltage across the capacitor starts at 0 and then increases exponentially until it reaches its maximum value of 10V whereas the plot of voltage across the resistor starts at 10V and decreases exponentially until it reaches its minimum value of 0V. The plot of current started at 0.5A which reaches close to 0A as the capacitor gets fully charged.

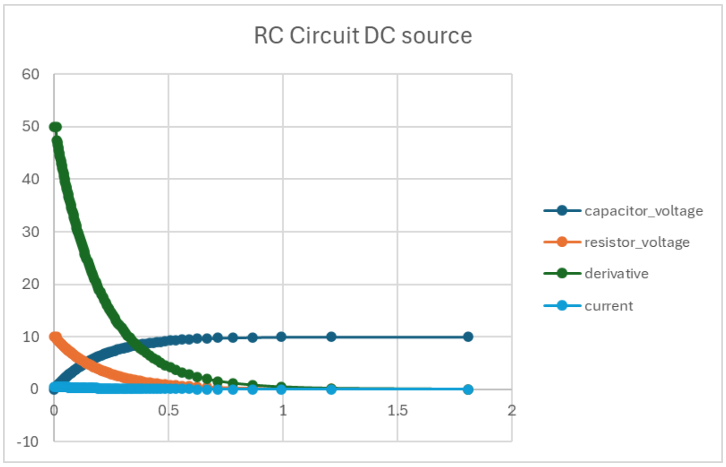

Figure 6: RC circuit simulation with DC source.

We then changed the Modelica code to replace the voltage source from constant 10V to 10*sin(t) and performed simulation and export of the model using Open-Modelica and then imported it into Cadmium for simulation using the QSS solver with a quantum size of 0.1. The value of quantum size was reduced to get better accuracy of the simulation results. Figure 7 shows the graph of values of input source voltage, the voltage across the resistor, the voltage across the capacitor, and the derivative of capacitor voltage observed during the simulation.

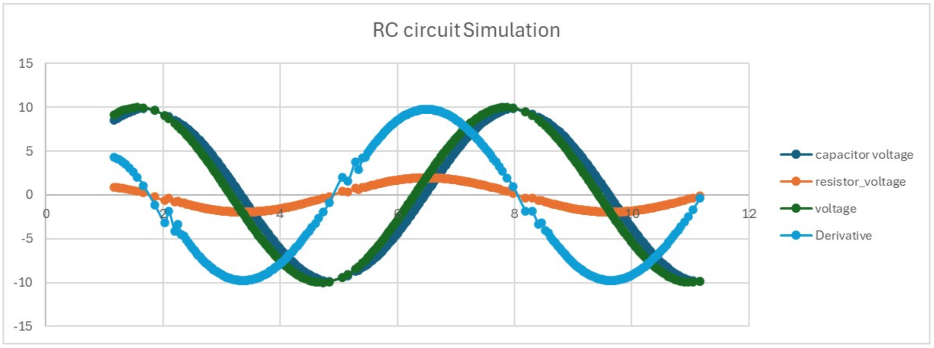

Figure 7: RC circuit simulation with AC source.

In the figure, we can observe that the plot of capacitor voltage was slightly delayed than the plot of input voltage with the peak voltage values at +10V and -10V. The plot of voltage across the resistors had 90 degrees of phase shift as compared to the input voltage with peak values at +2V and -2V, as expected.

5 CONCLUSION AND FUTURE WORK

In this paper, we discussed the definition and implementation of support for FMI development as well as the definition of a QSS solver for the simulation of a continuous time model in Cadmium. The model was developed and exported by using the Open-Modelica tool. The new C++-based library was developed by adding new classes into the modeling hierarchy to support the FMI standard in Cadmium. The simulation of the continuous-time equation was performed by the Quantized State System simulation method. We implemented the QSS solver with hysteresis. The Atomic model was developed to execute the functionality of the imported model. The modularity of DEVS is supported to smoothly integrate the model. We showed how to import and simulate Ordinary differential Equations in Cadmium which made use of the library developed for FMI import and the QSS solver and the results were validated using the simulation results of Open-Modelica.

The method can be adapted for multiple ODEs by calling the quantizer function multiple times and selecting the least value as a time advance value. The FMU import can be scaled to the import of multiple FMUs in the same model as different atomic models. The implementation still doesn't support the detection of FMU events for the import of a hybrid model. The method of FMI-import in the DEVS model shown in the paper needs full access to the FMU data for the simulation in the DEVS model. These kinds of experiments will be conducted in future iterations of this research. Similarly, we will expand support for the simulation of hybrid systems and the implementation of co-simulation on Cadmium. A technique for the detection of events from the imported model will be implemented and simulated using the same method introduced in this article. Support for the co-simulation APIs will be developed in the library which will allow importing the solver along with the model.

REFERENCES

[1] E. Kofman and S. Junco, "Quantized-state systems: a DEVS Approach for continuous system simulation," *Trans. Soc. Model. Simul. Int.*, vol. 18, no. 3, pp. 123–132, 2001.

[2] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation*. Academic press, 2000.

[3] T. Blochwitz *et al.*, "The functional mockup interface for tool independent exchange of simulation models," presented at the Proceedings of the 8th international Modelica conference, Linköping University Press, 2011, pp. 105–114.

[4] H. Elmqvist, S. E. Mattsson, and M. Otter, "Modelica: The new object-oriented modeling language," presented at the 12th European Simulation Multiconference, Manchester, UK, 1998.

[5] L. Belloli, D. Vicino, C. Ruiz-Martín, and G. Wainer, "Building DEVS models with the Cadmium tool," presented at the 2019 Winter Simulation Conference (WSC), IEEE, 2019, pp. 45–59.

[6] E. Kofman, J. S. Lee, and B. P. Zeigler, "DEVS representation of differential equation systems. review of recent advances," *Proc. ESS'01*, pp. 591–595, 2001.

[7] B. P. Zeigler and J. S. Lee, "Theory of quantized systems: formal basis for DEVS/HLA distributed simulation environment," presented at the Enabling Technology for Simulation Science II, SPIE, 1998, pp. 49–58.

[8] E. Kofman, "Discrete event based simulation and control of continuous systems," *PhD's Thesis Univ. Nac. Rosario Argent.*, 2003.

[9] M. H. Holmes, *Introduction to numerical methods in differential equations*. Springer, 2007.

[10] J. J. Benedetto and P. J. Ferreira, *Modern sampling theory: mathematics and applications*. Springer Science & Business Media, 2012.

[11] K. Gustafsson, M. Lundh, and G. Söderlind, "A PI stepsize control for the numerical solution of ordinary differential equations," *BIT Numer. Math.*, vol. 28, pp. 270–287, 1988.

[12] W. Borutzky, *Bond graph modelling of engineering systems*, vol. 103. Springer, 2011.

[13] U. M. Ascher and L. R. Petzold, *Computer methods for ordinary differential equations and differential-algebraic equations*. SIAM, 1998.

[14] F. E. Cellier and E. Kofman, *Continuous system simulation*. Springer Science & Business Media, 2006.

[15] F. Bergero, E. Kofman, and F. Cellier, "A novel parallelization technique for DEVS simulation of continuous and hybrid systems," *Simulation*, vol. 89, no. 6, pp. 663–683, 2013.

[16] P. Fritzson *et al.*, "The openmodelica modeling, simulation, and software development environment," *Simul. News Eur.*, vol. 44, pp. 8–16, 2005.

[17] T. Blockwitz *et al.*, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," presented at the Proceedings, 2012. [Online]. Available: https://elib.dlr.de/78486/1/otter2012-FMI20.pdf

[18] M. Tiller, *Introduction to physical modeling with Modelica*. Springer Science & Business Media, 2001.

[19] W. Müller and E. Widl, "Linking FMI-based components with discrete event systems," presented at the 2013 IEEE International Systems Conference (SysCon), IEEE, 2013, pp. 676–680.

[20] B. Camus *et al.*, “Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware,” *Simulation*, vol. 94, no. 12, pp. 1099–1127, 2018.

[21] X. Lin, “Co-simulation of Cyber-Physical Systems Using DEVS and Functional Mockup Units,” Arizona State University, 2021.

[22] S. Tripakis, “Bridging the semantic gap between heterogeneous modeling formalisms and FMI,” presented at the 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), IEEE, 2015, pp. 60–69.

[23] F. Bergero, A. Ranade, and F. Casella, “QSS and Multi-rate Simulation of Object-oriented Models,” presented at the Proceedings of the 7th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 2016, pp. 69–77.

[24] M. Wetter, T. S. Nouidui, D. Lorenzetti, E. A. Lee, and A. Roth, “Prototyping the next generation energyplus simulation engine,” presented at the Proceedings of the 3rd IBPSA Conference, Jeju island, South Korea, 2015, pp. 27–29.

[25] J. Fernández and E. Kofman, “A stand-alone quantized state system solver for continuous system simulation,” *SIMULATION*, vol. 90, no. 7, pp. 782–799, Jul. 2014.

[26] N. K. Sinha and G. P. Rao, *Identification of continuous-time systems: Methodology and computer implementation*, vol. 7. Springer Science & Business Media, 2012.

[27] D. J. Murray-Smith, *Continuous system simulation*. Springer Science & Business Media, 2012.

[28] D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, “Sequential PDEVS architecture,” presented at the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, 2015.

[29] B. Earle, K. Bjornson, C. Ruiz-Martin, and G. Wainer, “Development of a real-time DEVS kernel: RT-Cadmium,” presented at the 2020 Spring Simulation Conference (SpringSim), IEEE, 2020, pp. 1–12.

[30] F. Bergero, X. Floros, J. Fernández, E. Kofman, and F. E. Cellier, “Simulating Modelica models with a Stand–Alone Quantized State Systems Solver,” presented at the 9th International Modelica conference, 2012, pp. 237–246.

[31] F. Bergero, J. Fernández, E. Kofman, and M. Portapila, “Time discretization versus state quantization in the simulation of a one-dimensional advection–diffusion–reaction equation,” *Simulation*, vol. 92, no. 1, pp. 47–61, 2016.

[32] G. Migoni, E. Kofman, and F. Cellier, “Quantization–Based New Integration Methods for Stiff ODEs.,” *Quantization-Based New Integr. Methods Stiff ODEs*, 2010, [Online]. Available: https://www.researchgate.net/profile/Mario-Bortolotto/publication/257636805_Quantization-based_new_integration_methods_for_stiff_ordinary_differential_equations/links/600f43d145851553a06f9c80/Quantization-based-new-integration-methods-for-stiff-ordinary-differential-equations.pdf

[33] R. Cárdenas and G. Wainer, “Asymmetric Cell-DEVS models with the Cadmium simulator,” *Simul. Model. Pract. Theory*, vol. 121, p. 102649, Dec. 2022.

AUTHOR BIOGRAPHIES

RITVIK JOSHI is an M.A.Sc student at the Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada. He completed his Bachelor's in Electronics and Telecommunication Engineering from Savitribai Phule Pune University, India in 2019. His research interests include robotics, embedded systems, real-time systems, modeling, and simulation. His email address is ritvikjoshi@cmail.carleton.ca.

JAMES NUTARO is Group Lead for the Computational Systems Engineering & Cybernetics Group at Oak Ridge National Laboratory. He holds a Ph.D. in Computer Engineering from the University of Arizona. His research interests discrete event systems, systems modeling and simulation, and hybrid dynamic systems. His email address is nutarojj@ornl.gov.

BERNARD ZEIGLER is Professor Emeritus of Electrical and Computer Engineering at the University of Arizona (USA) and Chief Scientist of RTSync Corp. (USA). Dr. Zeigler is a Fellow of IEEE and SCS and received the INFORMS Lifetime Achievement Award. He is a co-director of the Arizona Center of Integrative Modeling and Simulation. His email address is zeigler@rtsync.com.

GABRIEL A. WAINER received the M.Sc. (1993) at the University of Buenos Aires, Argentina, and the Ph.D. (1998, with highest honors) at UBA/Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now a Full Professor. He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for Advanced Simulation and Visualization (V-Sim). He is an ACM Distinguished Speaker and a Fellow of SCS. His email address is gwainer@sce.carleton.ca.

DOOHWAN KIM is the founder and president of RTSync Corp., which specializes in Predictive Analytics and Model-Based System Engineering based on DEVS M&S technology. Dr. Kim has been involved in the design, development, and delivery of the advanced M&S solutions for highly complex real-world information science and engineering problems. He received his Ph.D. degree from the University of Arizona in 1996. His email address is dhkim@rtsync.com.