Co-Simulating Event-B and Continuous Models via FMI

Vitaly Savicks Michael Butler John Colley
Department of Electronics and Computer Science
University of Southampton
United Kingdom
vs2@ecs.soton.ac.uk mjb@ecs.soton.ac.uk J.L.Colley@ecs.soton.ac.uk

Keywords: Discrete-continuous systems, formal verification and validation, Event-B co-simulation, FMI master

Abstract

We present a generic co-simulation approach between discrete-event models, developed in the Event-B formal method, and continuous models, exported via the Functional Mock-up Interface for Co-simulation standard. The concept is implemented into a simulation extension for the Rodin platform, thus leveraging powerful capabilities of refinement-based modelling and deductive verification in Event-B while introducing a continuous-time aspect and simulation-based validation for the development of complex hybrid systems.

1. INTRODUCTION

Designing a complex hybrid system that consists of closely interacting discrete computing and intrinsically continuous physical processes is a challenging task [1]. A domain-specific tool can be effective for modelling and validating a subcomponent of a larger system, but is usually not sufficient when it comes to the design of a highly heterogeneous system, which requires seamless integration of a number of different tools and methods [2]. The Functional Mock-up Interface (FMI) standard was designed to solve this problem by providing a tool-independent open interface for the exchange and co-simulation of dynamic models [3].

Furthermore, as complexity increases, simulation-based analysis and bench-test solutions cannot fully guarantee system correctness. More rigorous verification techniques are essential, especially for the safety-critical domain [4]. Formal methods, based on the sound mathematical reasoning and theorem proving, offer a higher degree of confidence in the reliability and safety of developed systems, but are often criticised for the overall complexity, limited scalability and high demand on expertise [5, 6]. The development of advanced tools and methods tries to overcome these limitations, with the Event-B method and its open Rodin platform serving as a good example [7]. The latter offers automatic provers and supports a number of refinement, decomposition and instantiation techniques [8] that help to decrease model complexity and facilitate modular development. With the aid of numerous extensions to the platform, such as the requirements traceability [9], UML integration [10], model-checking [11], code generation [12, 13], etc., Event-B provides a comprehensive modelling framework for development of critical systems.

The discrete nature of the Event-B language and its simplified abstractions to facilitate the automatic proof, e.g. the absence of Reals, limits its capabilities in modelling continuous dynamics and timing properties. A number of proposals suggest ways to extend the language for hybrid systems [14–16]. In this work we follow a different path and offer an integration solution that is based on the co-modelling and co-simulation between the existing Event-B language, well-suited for discrete-event modelling and formal verification, and tool-independent continuous models of physical processes. Our concept utilises the FMI for Co-simulation standard [17] for continuous model integration with Event-B and is implemented into an extension to the Rodin platform that enables heterogeneous composition and simulation of Event-B and FMI models.

A number of integrated approaches for hybrid systems have been developed already. The notable examples are: the formalism of clocked data flow in Signal, co-simulated with continuous models in Simulink [18], the DESTECS project that separates the discrete-event and continuous-time aspects between the VDM and 20-sim respectively [19], the Ptolemy II environment that focuses on the heterogeneous hierarchical assembly of concurrent components and the use of well-known models of computation that govern their interaction [20]. The disadvantage of the mentioned approaches, in our opinion, is the limited integration with other modelling and verification technologies. A more generic co-modelling and co-simulation solution that is based on a tool-independent standard, such as the FMI, and coupled with a powerful proof-based formalism and a rich extensible toolset, such as Event-B/Rodin, aims to address these limitations.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to the Event-B formal method. Section 3 introduces the FMI standard architecture. In section 4 we present our concept of co-simulation, define the semantics of discrete and continuous simulation steps, and specify the simulation meta-model, API and a master algorithm. Section 5 demonstrates a proof of concept simulation experiment on a voltage distribution hybrid system. Conclusions and future directions are summarised in section 6.

2. EVENT-B

Event-B is a formal method for system-level modelling and analysis, inspired by the Action Systems [21] and B Method [22], and based on a mathematical notation of set theory and first-order logic to facilitate formal reasoning. System behaviour is modelled in Event-B as a collection of state variables and discrete conditional events that act on variables, while system properties are specified as invariants that can be formally verified by deductive proof. The key mechanism of refinement enables incremental modelling from abstraction towards implementation and splits the complex task of verification into manageable proofs. The Rodin toolset simplifies this task by providing automatic and interactive provers.

The main modelling constructs in Event-B are the static *contexts* that define *sets*, *constants* and *axioms*, and dynamic *machines*, which describe system properties (invariants to be verified) and the behaviour. The following is a representative Event-B machine that models a building access system [7]:

```
machine $m0$
sees $c0$
variables $registered, in, out$
invariants
  $registered \subseteq USER$ // set of registered users
  $in \subseteq registered$ // users inside the building
  $out \subseteq registered$ // users outside the building
  $in \cap out = \varnothing$ // cannot be simultaneously in&out
  $registered \subseteq in \cup out$ // reg. users are in or out
events
  $INITIALISATION \triangleq$
    $registered := \varnothing$
    $in := \varnothing$
    $out := \varnothing$
    end
  $Register \triangleq$
    any $u$
    where $u \in USER \setminus registered$
    then $registered := registered \cup \{u\}$
      $out := out \cup \{u\}$
    end
  $Enter \triangleq$
    any $u$
    where $u \in out$
    then $in := in \cup \{u\}$
      $out := out \setminus \{u\}$
    end
end
```

This abstract machine models a set of *registered* users, which can be either *in* or *out*. Two events model the functional requirements of the system. The *Register* event non-deterministically selects a non-registered user *u* and adds it to the registered/out set. The *Enter* event models entering the building by moving a registered/outside user to the set of inside users. The *INITIALISATON* event is a special Event-B event that initialises system variables to a valid state. The key invariants of this model are the last two.

At each refinement step we may introduce new variables and events (superposition refinement) or replace the abstract variables with the concrete ones (data refinement), thus incorporating more requirements into the model. A data refinement of the above example could be performed as follows:

```
machine $m1$
refines $m0$
sees $c1$
variables $registered, in, out, status$
invariants
  $status \in registered \rightarrow STATUS$ // user status
  $\forall u \cdot u \in registered \wedge status(u) = IN \Rightarrow u \in in$
  $\forall u \cdot u \in registered \wedge status(u) = OUT \Rightarrow u \in out$
events
  $INITIALISATION \triangleq$
    $registered := \varnothing$
    $status := \varnothing$
    end
  $Register \triangleq$
    refines $Register$
    any $u$
    where $u \in USER \setminus registered$
    then $registered := registered \cup \{u\}$
      $status(u) := OUT$
    end
  $Enter \triangleq$
    refines $Enter$
    any $u$
    where $status(u) = OUT$
    then $status := IN$
    end
end
```

The refinement step above represents a decision to replace the abstract sets *in* and *out* with a database-like *status* structure, in this case modelled by a function. A modeller must ensure that the refined machine/events imply the concrete model by relating concrete variables to their abstract counterparts. This is performed in Event-B by adding *gluing invariants*, two of which are defined in the above refined machine to relate the *status* to the sets *in* and *out*. The Rodin toolset provides model checking and automated proof features for verifying that machine events preserve invariants and for verifying the correctness of machine refinements.

3. FMI

The Functional Mock-up Interface[1] is an industrial tool-independent standard that defines cross-platform API for the exchange and co-simulation of dynamic models. It comes as a set of C header files to be implemented by an individual

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

model and a `modelDescription.xml` file schema for describing model-specific properties and state variables [23]. The implemented model code can be compiled into a dynamic/shared library for the target platform and bundled with the model description file into a Functional Mock-up Unit (FMU), which is essentially a `.zip` archive, ready to be used for modelling and simulation in an FMI-compliant tool.

The standard is split into two parts: *Model Exchange*, for shipping a model as an input/output block, which can be utilised by other environments; and *Co-Simulation*, for coupling models and simulators in a co-simulation environment, where each subsystem, called *Slave*, is solved by its individual solver. The data exchange between the Slaves and the synchronisation of their solvers is coordinated by a simulation algorithm, called *Master*.

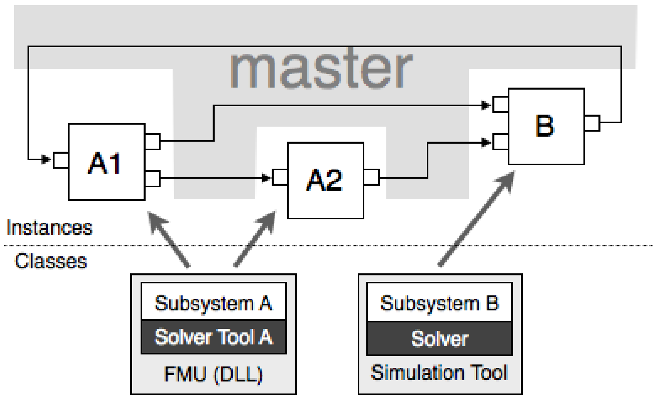

Figure 1. FMI master-slave architecture

A slave can represent either an exported dynamic-link library (*Subsystem A* in Figure 1) that may be instantiated multiple times (*A1* and *A2*) or a coupled simulation tool that simulates a model (*Subsystem B*). A coupled subsystem can be either continuous in time (described by differential equations) or discrete (difference equations), and can be represented as a block with inputs, outputs and internal (state) variables. Subsystem variables, their causality (input, output, internal) and type (Real, Integer, Boolean, String), along with the information about the model, solver and simulation capabilities are described in a slave-specific model description file.

Physical connections between subsystems are represented by mathematical coupling conditions among their inputs and outputs [24]. This information can be effectively encoded in a component connection graph and used for the data exchange aspect of the FMI for Co-Simulation. The synchronisation of the simulation from time $tc_0 = t_{start}$ to $tc_N = t_{stop}$ happens in communication steps $tc_i \rightarrow tc_{i+1}$, and is the responsibility of the master algorithm. The FMI for Co-Simulation supports not only fixed-step algorithms, but also more sophisticated approaches that adapt the step size to the solution behaviour, use higher order signal extrapolation to approximate subsystem inputs, or handle simulation steps sequentially such that the intermediate results from the first subsystems may be used to improve the approximation of subsystem inputs in the later stages of the communication step [17, 25, 26]. The standard is designed to support a very general class of algorithms and gives the guidelines on the basic implementation. However, it does not define the master algorithm itself.

4. CO-SIMULATION

This section presents the concept of a generic co-simulation framework for Event-B, starting with a description of the simulation step semantics of discrete/continuous components, followed by a simulation meta-model and a basic master algorithm. The idea has been implemented into a prototype extension for the Rodin platform using the standard Eclipse technologies[2], ProB animator[3] and JFMI Java implementation of the FMI library[4].

4.1. Concept

Our concept of co-simulation between Event-B and continuous models is based on the master-slave architecture of the FMI for Co-Simulation v1.0 standard [17], which provides an abstract fixed-step master algorithm that we have utilised in our prototype tool. According to the FMI architecture the simulation process is divided into simulation steps, whose boundaries serve as synchronisation and data exchange points. We further distinguish the simulation step of a discrete component, represented by an Event-B machine, and a continuous component that denotes an FMU.

The semantics of a continuous step (we call it *CStep*) is defined within the FMI standard as a function call to its simulator, which is responsible for simulating the underlying model for a specified period of time.

The discrete simulation step (*DStep*) is represented either by a single Event-B event or a number of events, executed sequentially according to the Event-B semantics. Instead of explicitly specifying the sequence of events that constitute a simulation step we have introduced the notion of a *Wait* event, which denotes the end of the step. Essentially, Wait must be the only executable event(s) at synchronisation points, whereas the sequence of events within a step can be arbitrary and is defined by individual model's logic. This provides a generic simulation step solution and a flexible model of refinement of the discrete step.

To synchronise the simulation the master keeps the record of the global simulation time. The state of an individual slave can thus be defined as a function:

$$F : Time \rightarrow V \tag{1}$$

[2]http://projects.eclipse.org/projects/modeling
[3]http://www.stups.uni-duesseldorf.de/ProB
[4]http://ptolemy.eecs.berkeley.edu/java/jfmi/

where V is the state of the slave's internal variables. The evolution of each variable (and therefore each slave) over time can be represented on a graph:

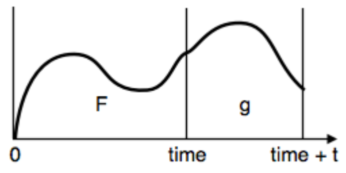

Figure 2. The state of a slave over time

where g is a state function defined over time interval $time \dots time + t$. The master only synchronises simulation at fixed points in time, when it exchanges data between connected slaves and simulates them to the next step. If we assume that t equals the step period of the master, the simulation semantics of each slave can be formally defined using Event-B notation as follows:

machine C
variables $F, time$
event $CStep =$
any i, t, g
where
 $g \in [time \dots time + t] \to V$
 $g(time) = F(time)$
 $P(g, i, F, time, t)$
then
 $time := time + t$
 $F := F \cup g$

where parameter i is slave inputs and P is model properties, or properties that g must satisfy. This formal model specifies the semantics of continuous slaves, as it depends on time and is based on continuous function F. For the discrete slaves of Event-B we can derive a simpler definition that depends on input and internal variables:

machine D
var V, O
event $DStep =$
any i
where
 $i \in T$
then
 $V, O := S(V, O, i)$

where i is the input, O is internal variables that are also outputs, and S is a discrete state function. As global time is absent in Event-B models, to synchronise them with other slaves the master uses the Wait event as an indication of the end of discrete simulation step. The Wait event must be an existing Event-B machine event, which pauses further execution at the end of a simulation cycle. The reading of inputs, on the other hand, precedes the simulation step and therefore must be performed by a single or multiple enabled events, which we denote as *ReadInput* events.

The above concepts are illustrated on a water tank example that consists of a leaking water tank and a controller that controls the input valve to maintain the desired water level:

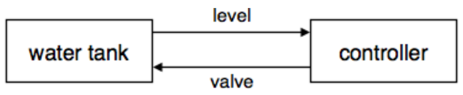

Figure 3. Controlled water tank model

The plant and controller can be modelled as a continuous and discrete slave, accordingly, that exchange a continuous signal *level* and a discrete signal *valve*. The control flow of the simulation can be shown as a state chart [27] in Figure 4.

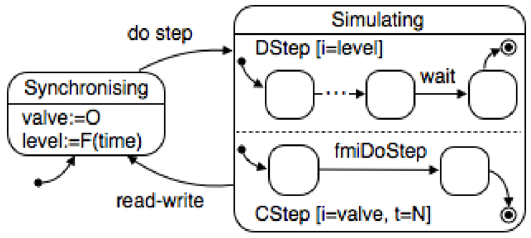

Figure 4. Control flow of a discrete-continuous co-simulation

The state chart shows that simulation steps of the controller (DStep) and the tank (CSep) may be executed in parallel. The DStep requires execution of a number of arbitrary events until a Wait event is enabled. The CStep directly maps to an `fmiDoStep` method of the FMI. Additionally, a read-write phase is required prior to the simulation step to exchange the signal data between two components.

4.2. Example Mapping

To demonstrate how Event-B machines map to Event-B components, used in the simulation, we show a simple refinement of a water tank controller model in Event-B:

machine $tankController0$
variables $valve$
events
 $SwitchOn \mathrel{\hat{=}}$ **any** l **where** $l < LT$ **then** $valve := on$ **end**
 $NoSwitch \mathrel{\hat{=}}$ **any** l **where** $l \geq LT \land l \leq HT$ **then** $skip$ **end**
 $SwitchOff \mathrel{\hat{=}}$ **any** l **where** $l > HT$ **then** $valve := off$ **end**
end

In this abstract machine all three events are ReadInput events and Wait events. Only one of these events gets executed in a simulation cycle, depending on the value of l that is an input signal from the plant.

When $tankController0$ is refined to introduce multiple sequential discrete steps, then the ReadInput and Wait events become distinct. $tankController0$ could be refined as a state machine as follows:

machine $tankController1$ **refines** $tankController0$
variables $valve, level, state$
events
$ReadLevel \triangleq$ **any** l **where** $state = 0$ **then** $level := l$ **end**
$DecideOn \triangleq$ **where** $state = 1 \wedge level < LT$ **then** $state := 2$ **end**
$DecideSkip \triangleq$ **where** $state = 1 \wedge level \geq LT \wedge level \leq HT$
then $state := 3$ **end**
$DecideOff \triangleq$ **where** $state = 1 \wedge level > HT$ **then** $state := 4$ **end**
$SwitchOn$ **refines** $SwitchOn \triangleq$
witness $l = level$ **where** $state = 2$ **then** $valve := on$ **end**
$NoSwitch$ **refines** $NoSwitch \triangleq$
witness $l = level$ **where** $state = 3$ **then** $skip$ **end**
$SwitchOff$ **refines** $SwitchOff \triangleq$
witness $l = level$ **where** $state = 4$ **then** $valve := off$ **end**
end

In the refinement *ReadLevel* is a ReadInput event and *SwitchOn*, *NoSwitch* and *SwitchOff* are Wait events. This flexibility of indicating multiple and/or same events as ReadInput and Wait events enables the refinement of control events and, most importantly, verification and co-simulation of Event-B components from the early stage of development, which is crucial for safety-critical systems.

4.3. Meta-model and Semantics

FMI for Co-Simulation specifies the generic interface routines, or the Application Programming Interface (API), for the communication between the master and the slaves [17]. Following the same idea we have defined an API for a generic class *Component*, which is specialised by Event-B and FMI components that denote Event-B machines and FMU models, accordingly. This genericity greatly simplifies the master and allows us to extend co-simulation to other types of components without modifying the master algorithm.

All component classes and other simulation constructs, such as component ports, for reading input signals and producing output signals, or connectors, for connecting multiple components together, have been defined in a single meta-model[5] that encodes the simulation semantics and is used for constructing individual co-model graphs.

FMU Components implement our simulation API directly via callback functions to the underlying FMI API. In order to operate FMI models from Rodin we have implemented an interface in the ProB Core using the JFMI Java wrapper.

Event-B Components implement the API according to the discrete step semantics that has been defined earlier. For example, the `doStep` method of the API executes a sequence of the Event-B machine's enabled events non-deterministically until one of the Wait events is enabled. The `stepPeriod` property of the Event-B Component specifies the time duration of a single simulation step for that component. The data exchange is performed by `writeOutputs` and `readInputs` API methods. The latter executes Event-B ReadInput event(s), passing them the values obtained from the input ports. The `writeOutputs` method reads the values of the corresponding Event-B variables directly and writes them to the output ports.

[5]In model-driven engineering a meta-model is a model of a modelling language, i.e. a specification of the rules and constructs for creating semantic models [28].

4.4. Master Algorithm

Our simulation master algorithm is designed to comply with the FMI standard (though it is not part of the standard itself), i.e. we have developed it to reflect the recommended use of the FMI API. The outline of the algorithm is as follows:

1. Instantiate all slaves:
   `Component.instantiate())`
2. Initialise all slaves:
   `Component.initialise(startTime, stopTime)`
3. Set global time to a start time and begin the simulation loop
4. For each slave, write all outputs:
   `Component.writeOutputs())`
5. For each slave, read all inputs:
   `Component.readInputs())`
6. Perform simulation step:
   `Component.doStep(time, stepSize))`
7. Increase time by step size; if time has reached the stop time then stop, otherwise go back to step 4
8. Terminate slaves:
   `Component.terminate())`

The master uses the generic component API of the meta-model from 4.3., therefore it does not rely on a particular implementation of the individual component type, either Event-B or FMU, and making it possible to extend co-simulation capabilities by introducing other types of components without the need to modify the master itself.

It is worth noting that the step size is treated differently by two types of components, in particular, by the Event-B component, which has an attribute for the step size, set by the modeller. The attribute defines the duration of the simulation step and characterises its component at the simulation master level, i.e. an Event-B machine itself does not need to be timed (although it can be, in which case additional mechanisms for reading and writing time to Event-B are required). A reasonable choice of the step size value should be no smaller than the simulation master step size, otherwise it may be missed by the master algorithm.

5. VOLTAGE DISTRIBUTION CONTROL

To validate our approach we have conducted a number of simulation experiments, one of which has been taken from the power systems domain in order to illustrate how the co-simulation can be used in modelling and verification of smart grid systems[6]. A standard electric power system consists of the generating stations (power plants, wind turbines, solar farms), high-voltage transmission lines, distribution networks and residential/commercial consumption loads [29]. The final distribution segment must step down the distribution voltage to a residential voltage that is safe for use by general consumers. The task of stepping the voltage up/down is performed by a transformer with a tap changer that allows the winding ratio between the primary (input) and secondary (output) voltage to be varied. The latest generation of the digitally controlled On-Load Tap Changers (OLTC) allows automatic voltage regulation by varying the transformer ratio under load without interruption.

In this experiment we are concerned with modelling the final distribution segment of an electric power system, in which a distribution voltage of 11kV is converted by an OLTC transformer to a consumption voltage of 230V. The system goal is to maintain the reference voltage of 230V within a predefined deadband (safe range) under any load conditions. One of the means of achieving this is by monitoring the voltage under load and controlling the OLTC position. The continuous part of the system has been modelled in Modelica, which is an object-oriented equation-based language, suitable for describing physical processes in a structural way [30]. The resulting model, shown in Figure 5, was constructed from components of the PowerSystems Modelica library[7].

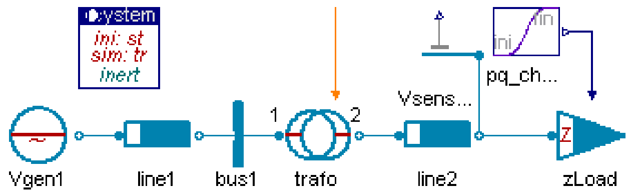

Figure 5. Distribution voltage control system in Modelica

The model consists of a constant voltage source *Vgen1* that represents a primary distribution voltage, two power lines split by an OLTC transformer *trafo*, and a load *zLoad* that represents a residential area. The load sinusoidally increases five times of the nominal over the period of 30 seconds, which leads to a corresponding voltage drop. To regulate the voltage an input control signal can switch the secondary tap of the transformer by providing an index of the tap position. There are 21 positions defined in the transformer, with a 0.2 ratio step between the two consecutive positions. A monitored voltage from the voltage sensor *Vsensor1* is sent back to the controller.

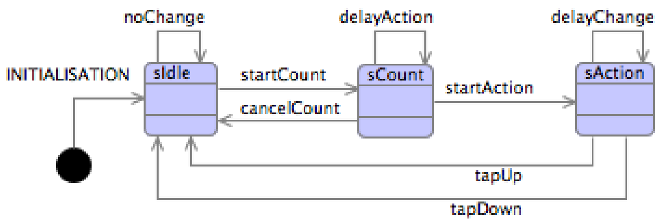

Figure 6. Event-B state machine of the OLTC controller

The OLTC controller has been modelled in Event-B as a state machine (Figure 6) according to [31]. The state machine consists of three states:

- *sIdle*, denoting a normal operation mode where a monitored voltage is within the deadband. The *noChange* event indicates the end of simulation step in this mode.
- *sCount*, denoting a state where the voltage is outside of the deadband, but no tap change action is yet taken. This mode models a detection delay of the OLTC that monitors the voltage for a certain amount of time in case it goes back to normal (transition *cancelCount*) before taking any action. The delay is modelled in Event-B by a decreasing counter variable *dCounter*. The *delayAction* event is the Wait event in this mode.
- *sAction*, which models a mechanical delay of the tap changer after detection delay expires (transition *startAction*). After another counter variable *mCounter*, involved in this state, reaches zero a corresponding tap step up/down action is performed (transition *tapUp* or *tapDown*, respectively). The Wait event in this mode is *delayChange*.

The co-simulation settings of the OLTC/controller had a detection delay set to 5 s, a mechanical delay of 1 s and a deadband of 2V. The step period of the controller was set to 0.1 s and the simulation run for 50 s to observe the voltage drop and the reaction of the tap changer, shown in Figure 7.

The simulation results demonstrate that the controller detects a deviation from the deadband ($vNorm < 229$V) at t = 23 s and switches to the *sCount* state. As voltage continues to stay outside the acceptance range for 5 s an action is taken at t = 28 s and is completed after a mechanical delay of 1 s. The tap position changes from 11 (middle position, denoting the nominal ratio) to 12, i.e. a tap up is performed to step up the secondary voltage. As a result the voltage jumps to 232V at

[6]The output of this works is used in a smart grid case study of the ADVANCE project: `http://www.advance-ict.eu`

[7]`https://github.com/modelica/PowerSystems`

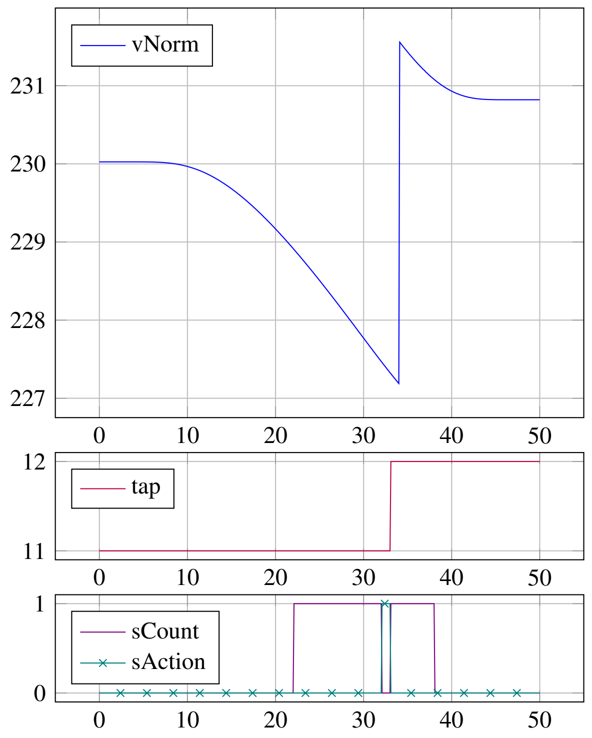

Figure 7. Co-simulation results of the OLTP voltage control (simulation time = 50 s, step size = 0.1 s)

t = 29 s and becomes outside of the range, but goes back to normal as the load continues to increase.

A more interesting scenario we would like to model in the future is a number of different factors affecting the voltage (line drop, distribution generation, etc.), as well as a more complex model of the residential load and an intelligent OLTC control algorithm that minimises the number of tap changes to minimise the wear of expensive equipment.

6. CONCLUSIONS

The proposed co-simulation approach and the corresponding tool for co-simulating Event-B models and FMUs is still under development and requires further research. However, the obtained experimental results on a number of case studies from avionics and power systems domain have clearly demonstrated the feasibility of developing a framework for a general class of hybrid systems that not only facilitates tool-independent model composition and simulation, but most importantly integrates formal modelling and verification of discrete-event systems, such as controllers, and co-simulation of the latter with continuous-time physical models of environment. With respect to the technologies used in this work, Event-B provides the ability to derive a correct implementation of a discrete controller through refinement and Rodin verification support. Co-simulation allows us to do closed-loop validation of an Event-B model of a controller with a continuous model of the plant being controlled. Among the potential benefits of such framework is a greater degree of reliability and safety of developed systems through the application of rigorous analysis, a deeper understanding of the computation-environment interaction and a stimulus for further integration and co-development between domain-specific tools. Our future work includes a stronger formalisation of the co-simulation semantics, development of an adaptive master algorithm and performance/scalability analysis of our tool on larger case studies.

7. ACKNOWLEDGEMENT

This work is part of the ADVANCE Project (Advanced Design and Verification Environment for Cyber-physical System Engineering) funded by the European Commission (http://www.advance-ict.eu).

REFERENCES

[1] Edward A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008. Invited Paper.

[2] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.

[3] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The Functional Mockup Interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

[4] Stefania Gnesi and Tiziana Margaria. *Formal Methods for Industrial Critical Systems*. Wiley Online Library, 2013.

[5] S. Liu and R. Adams. Limitations of formal methods and an approach to improvement. In *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, pages 498–507. IEEE, 1995.

[6] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.

[7] J.R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[8] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1):1–28, 2007.

[9] Michael Jastram. ProR, an open source platform for requirements engineering based on RIF. 2010.

[10] C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. 2008.

[11] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[12] Steve Wright. Automatic generation of C from Event-B. In *Workshop on integration of model-based formal methods and tools*. Citeseer, 2009.

[13] Andrew Edmunds and Michael Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. 2011.

[14] Jean-Raymond Abrial Wen Su and Huibiao Zhu. Complementary methodologies for developing hybrid systems with Event-B, 2012.

[15] Jean-Raymond Abrial, Wen Su, and Huibiao Zhu. Formalizing hybrid systems with Event-B. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 178–193. Springer, 2012.

[16] Richard Banach and Michael Butler. A hybrid Event-B study of lane centering. In *Complex Systems Design & Management*, pages 97–111. Springer, 2014.

[17] MODELISAR. Functional Mock-up Interface for Co-Simulation, Version 1.0. `https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_CoSimulation_v1.0.pdf`, October 2010.

[18] S. Tudoret, S. Nadjm-Tehrani, A. Benveniste, and J.E. Strömberg. Co-simulation of hybrid systems: Signal-Simulink. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 623–639. Springer, 2000.

[19] J. Fitzgerald, P. Larsen, K. Pierce, M. Verhoef, and S. Wolff. Collaborative modelling and co-simulation in the development of dependable embedded systems. In *Integrated Formal Methods*, pages 12–26. Springer, 2010.

[20] Christopher Brooks, Edward A Lee, Xiaojun Liu, Yang Zhao, Haiyang Zheng, Shuvra S Bhattacharyya, Elaine Cheong, Mudit Goel, Bart Kienhuis, Jie Liu, et al. Ptolemy II: Heterogeneous concurrent modeling and design in Java. 2005.

[21] Ralph-JR Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

[22] J. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Sørensen. The B-method. In *VDM'91 Formal Software Development Methods*, pages 398–405. Springer, 1991.

[23] Torsten Blochwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference, Munich*, 2012.

[24] R Kübler and W Schiehlen. Two methods of simulator coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2):93–113, 2000.

[25] Jens Bastian, Christoph Clauß, Susann Wolf, and Peter Schneider. Master for co-simulation using FMI. In *8th International Modelica Conference. Dresden*, 2011.

[26] Tom Schierz, Martin Arnold, and Christoph Clauß. Co-simulation with communication step size control in an FMI compatible master algorithm. In *9th International Modelica Conference. Munich*, 2012.

[27] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[28] Edwin Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, 2003.

[29] Birron Mathew Weedy, Brian John Cory, N Jenkins, JB Ekanayake, and G Strbac. *Electric power systems*. John Wiley & Sons, 2012.

[30] Modelica Association et al. Modelica – a unified object-oriented language for physical systems modeling. *Language Specification, Version*, 2, 2005.

[31] Mohammad Moradzadeh and René Boel. A hybrid framework for coordinated voltage control of power systems. In *IPEC, 2010 Conference Proceedings*, pages 304–309. IEEE, 2010.