A Flexible Framework for FMI-based Co-Simulation of Human-Centred Cyber-Physical Systems

Maurizio Palmieri[2,1], Cinzia Bernardeschi[1], and Paolo Masci[3]

[1] Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

[2] Dipartimento di Ingegneria dell'Informazione, University of Florence, Italy

[3] HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. This paper presents our on-going work on developing a flexible framework for formal co-simulation of human-centred cyber-physical systems. The framework builds on and extends an existing prototyping toolkit, adding novel functionalities for automatic generation of user interface prototypes equipped with a standard FMI-2 co-simulation interface. The framework is developed in JavaScript, and uses a flexible templating mechanism for converting stand-alone device prototypes into Functional Mockup Units (FMUs) capable of exchanging commands and data with any FMI-compliant co-simulation engine. Two concrete examples are presented to demonstrate the capabilities of the framework.

1 Introduction

Human-centered Cyber-Physical Systems (CPS) are complex systems that integrate human operators, digital controllers, and the physical world. An example is a self-driving car where an advanced driver assistance system automatically adjusts the speed and navigation of the car based on inputs from sensors, and the driver can take over control of the car at any point in time, e.g., by pressing the brake or accelerator pedal.

Model-based simulation technologies applied at the early stages of system design allow developers to gain additional confidence that the system behaves as expected. To produce accurate results in model-based analysis of CPS, developers often need to use co-simulation techniques, i.e., integrated simulation of different sub-systems, each modelled and simulated with the most appropriate tool (e.g., logic-based models for digital controllers, and continuous models based on differential equations for the physical part of the system).

To date, the research community has devoted most of its effort to the development of tools for co-simulation of cyber and physical components of CPS. Relatively little attention has been dedicated to developing tool support to assess the design of the human-machine interface of CPS, even though human-CPS interaction is often a critical aspect of the system, e.g., see the recent accidents

involving self-driving cars [16, 15], where the design of the car dashboard exceeded the driver's abilities and performance when the driver needed to take over control of the car because of an emergency situation.

Contribution. We present a framework designed to support modelling and co-simulation of the user interface of a CPS. The framework builds on and extends PVSio-web [12], a prototyping toolkit for model-based analysis of human-machine interfaces. We extend PVSio-web to introduce support of automatic generation of user interface prototypes equipped with a standard FMI-2 co-simulation interface. Our framework is developed in JavaScript, and uses a flexible templating mechanism to convert stand-alone device prototypes into Functional Mockup Units (FMUs) capable of exchanging commands and data with any FMI-compliant co-simulation engine. Two example co-simulations of CPS are presented to illustrate the features and utility of the framework.

Structure. Section 2 illustrates background tools and concepts used in the work. Section 3 presents the code for automatically generate an FMU implementing a device prototype previously built with PVSio-web. Section 4 shows two different example applications of our work. Section 5 presents related work on co-simulation of CPS. Finally section 6 concludes the paper.

2 Background

In this section we provide details on the two main technologies used in this work, namely PVSio-web and Functional Mockup Interface (FMI). PVSio-web because is a flexible tool for simulation of graphic user interfaces of CPS based on an Higher Order Language (PVS) and FMI is an emerging standard for co-simulation of CPS.

PVSio-web. PVSio-web [12] is a toolkit for prototyping and analysis of interactive (human-centred) systems. An example prototype developed with PVSio-web is shown in the upper part of Figures 2 and 4. Each PVSio-web prototype consists of two parts: a back-end defining the behaviour of the system; and a graphical front-end defining the visual appearance of the system. The behaviour of the system is specified as an executable formal specification in PVS [17]. The visual appearance of the system is an interactive picture of the real system. Web technologies (HTML5 & JavaScript) are used to create hotspot areas over the picture, and link input and output widgets to the PVS specification. Input widgets translate user actions over buttons into PVS expressions to be evaluated in PVSio [14], the animation component of PVS, to compute the system response. Output widgets mirror the value of state attributes of the PVS model using graphic elements reproducing the look&feel of the real system in the corresponding state. A library of widgets is provided by PVSio-web that includes common interactive elements of a system (buttons, digital displays, gauges, etc.).

Functional Mockup Interface. The Functional Mockup Interface (FMI) [3] is a tool-independent standard for co-simulation of dynamic models. Co-simulation is performed by a number of *Functional Mockup Units* (FMUs), each responsible

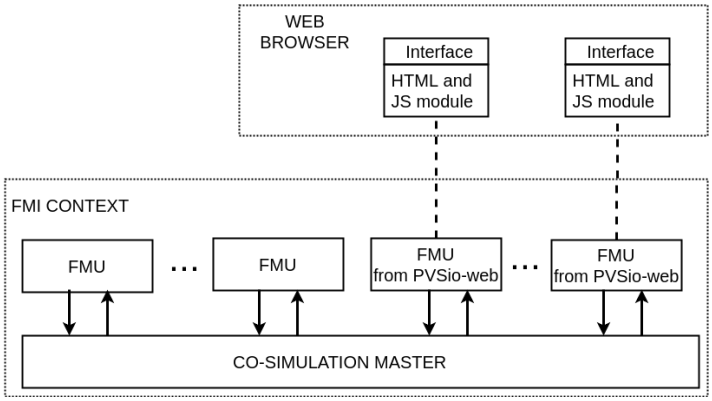

Fig. 1: FMI architecture including FMU generated from PVSio-web

for simulating a single model in the native formalism and execution environment of the tool used to create the model. An FMU may carry a whole simulation environment, or just information needed by an FMI-compliant host environment to simulate the model contained in the FMU. An FMI-compliant host environment provides a *master* program that communicates with other FMUs acting as *slaves*. The APIs of each FMU include: initialisation functions; a function `fmi2DoStep` that triggers one simulation step; and functions to exchange data, including getter and setter functions `fmi2Get`$<TYPE>$ and `fmi2Set`$<TYPE>$, where $<TYPE>$ is a concrete type name, e.g., *Integer* or *Real*.

3 Our Framework

Our framework allows developers to extend stand-alone PVSio-web prototypes with an FMI-2 compliant co-simulation interface. That is, given a prototype developed with PVSio-web, our framework generates an FMU that includes:

- The PVS model of the prototype specifying the behaviour of the prototype;
- The PVSio environment necessary for executing the PVS model;
- The XML description file used in FMI-based co-simulations to specify static information of the model (such as the list of variables);
- C code implementing the APIs of the FMU necessary for exchanging data and commands with other FMUs;
- C code implementing a web server necessary to communicate with the graphical front-end of the PVSio-web prototype;
- An external module for executing the graphical front-end of the prototype in a web browser.

The overall architecture of a co-simulation where one or more FMUs are PVSio-web prototypes is shown in Figure 1 (additional details will be provided further below, in subsection 3.1).

3.1 Communication between FMU and the prototype interface

FMUs encapsulating PVSio-web prototypes use a WebSocket to exchange data and commands with the graphical front-end of the prototype (see Figure 1). That is, the graphical front-end communicates only with the FMU, and does not interact directly with the co-simulation engine. This design choice promotes a modular architecture of the FMU, and enables *hot swapping* of different look&feel of the device without restarting the co-simulation — this is useful, e.g., when using the prototypes for design exploration. In the following we briefly describe the interaction between the FMU and the user interface of the prototype.

When the user performs an action on the graphical user interface of a PVSio-web prototype, the JavaScript module sends a message to the FMU with information about the action that has been performed (e.g., button x has been clicked). Every time the co-simulation master invokes a simulation step, the FMU checks if a new message has been received from the user interface (line 14 of Listing 1.4). If a message has been received, the FMU executes the user command first, and then a simulation step. After the execution of the action received from the user interface, the FMU replies to the user interface, via the same websocket connection, sending the updated state of the system.

The graphical user interface of the PVSio-web prototype is detached from the FMU. In case no user action is performed on the graphical user interface, consistency between the state of the co-simulation and feedback of the user interface is supported by an action *refresh* automatically sent by the front-end at each co-simulation step.

3.2 The APIs of our framework

The APIs provided by our framework include functionalities for generating the XML description file and the C code implementing the standard FMI functions necessary to extend a PVSio-web prototype with an FMI interface. The APIs are implemented in JavaScript, and the principal API function is `create_FMU`. An example use of the `create_FMU` function is as follows:

```
1 fmi_module.create_FMU("line_following_robot", {
2   fmi: [ { name : "gear", type :"string", variability: "discrete",
3            scope:"local", value:"0" },
4          ... ],
5   init: "init_LFR",
6   tick: "tick"
7 });
```

The first argument (`line_following_robot`) is the name of the FMU. The second argument is an object with three attributes:

- `fmi`: an array specifying the characteristics (name, type, variability, etc.) of the co-simulation variables;
- `init`: the name of the function in the PVS model for initializing the PVSio-web prototype;
- `tick`: the name of the function in the PVS model for advancing time.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <fmiModelDescription fmiVersion="2.0" modelName="{{modelName}}" ...>
3    <CoSimulation modelIdentifier="{{modelName}}"
4                  canHandleVariableCommunicationStepSize="false" ...>
5    </CoSimulation>
6    <LogCategories><Category name="logAll" /> ... </LogCategories>
7    <ModelVariables>{{#each variables}}{{#if fmi}}
8      <ScalarVariable name="{{name}}"
9                      valueReference="{{fmi.valueReference}}"
10                     causality="{{fmi.causality}}"
11                     variability="{{fmi.variability}}" >
12       <{{fmi.descriptor}} {{#if input}}start="{{value}}"{{/if}}
13                           {{#if parameter}}start="{{value}}"{{/if}} />
14     </ScalarVariable>{{/if}}{{/each}}
15   </ModelVariables>
16   <ModelStructure> ... </ModelStructure>
17 </fmiModelDescription>
```

Listing 1.1: Handlebars template for generating the XML description file.

The Handlebars[4] engine is used for generating the source code of `create_FMU` and other functions. The engine supports semantic templates with *parameters* and *helper functions*. Template parameters are instantiated at run time, using information contained in JSON objects. Helper function enable conditional compilation and iteration over arrays. The advantage of using semantic templates is that the structure of the source code can be inspected in the template, e.g., to check the correctness of syntax and semantics of the code to be generated. This makes it easier for developers to update the template when necessary, e.g., to adapt code generation to future versions of the FMI standard or to different platforms. We used the same approach in [13] for generating MISRA-C code from diagrams based on the state-charts notation. Details of the Handlebars templates developed for XML and C code generation are in the following subsections.

3.3 Generation of the XML description file

Relevant fragments of the Handlebars template for generating the XML description file of an FMU are shown in Listing 1.1. Template parameters are characterised by unique identifiers and are adorned with curly braces. An example parameter in Listing 1.1 is {{`modelName`}}, which represents the name of the model described by the XML file. This and other template parameters are instantiated by invoking the Handlebars compilation engine with a JSON object whose attributes specify the actual values of those parameters. Helper functions {{`if`}} and {{`each`}} are used in the Handlebars template to perform conditional compilation (e.g., see lines 12-13 in Listing 1.1) and iteration over arrays (e.g., see lines 7-14 in Listing 1.1).

An example XML file generated using the template is shown in Listing 1.2. The first part of the file provides general information about the FMU (e.g., model name, author, etc.) and information about co-simulation options supported by the FMU (e.g., step-size). The main body of the file contains information about

[4] https://handlebarsjs.com

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription fmiVersion="2.0" modelName="line_follower_robot"...>
  <CoSimulation
      modelIdentifier="line_follower_robot"
      canHandleVariableCommunicationStepSize="false"
      ...>
  </CoSimulation>
  <LogCategories><Category name="logAll" /> ... </LogCategories>
  <ModelVariables>
    <ScalarVariable name="gear" valueReference="1"
                    causality="local" variability="discrete">
                    <String /></ScalarVariable>
                    ...
    <ScalarVariable name="lightSensors_right" valueReference="10"
                    causality="input" variability="continuous">
                    <Real start="0" /></ScalarVariable>
    <ScalarVariable name="motorSpeed_left" valueReference="11"
                    causality="output" variability="discrete">
                    <Real /></ScalarVariable>
                    ...
  </ModelVariables>
  <ModelStructure> ... </ModelStructure>
</fmiModelDescription>
```

Listing 1.2: Example XML description file generated with our template.

variables used in the co-simulation, specified according to the format required by the FMI standard:

- `valueReference` is the buffer index where the value of the variable is stored;
- `causality` defines if the variable is input (i.e., received from another FMU), output (i.e., sent to another FMU), local (i.e., the variable is only used within the FMU), or if it is a parameter of the FMU;
- `variability` defines how the variable changes over time (i.e., discrete time or continuous time), or if the variable has a constant value.

3.4 Generation of the C code implementing the APIs of the FMU

The Handlebars template for generating the FMU of a PVSio-web prototype includes the definition of the standard FMI functions for exchanging data between FMUs (`fmi2DoStep`, `fmi2Instantiate`, etc.), and additional interface functions necessary to enable communication between front-end and back-end of the PVSio-web prototype. The graphical front-end is implemented in HTML5 & JavaScript, and executed in a web browser. The back-end is embedded in the FMU and executed within a web server encapsulating the PVSio animation environment. As an example, a snippet of the Handlebars template for generating function `fmi2DoStep` is shown in Listing 1.3. The function is used by a co-simulation master to trigger the execution of a simulation step in the FMU. It includes four arguments:

- `fmi2Component` is the FMU;
- `currentCommunicationPoint` is the current simulation time;
- `communicationStepSize` is the simulation step;
- `noSetFMUStatePriorToCurrentPoint` is a boolean that specifies if the master can revert the state of the FMU back to a prior simulationn time.

```
fmi2Status fmi2DoStep(fmi2Component c,
                      fmi2Real currentCommunicationPoint,
                      fmi2Real communicationStepSize,
                      fmi2Boolean noSetFMUStatePriorToCurrentPoint) {
    doStep();
    return fmi2OK;
}
```

Listing 1.3: Snippet of Handlebars template for `fmi2DoStep`.

```
1  void doStep() {
2    // read input variables
3    {{#each variables}}{{#if fmi}}{{#if input}}
4      {{#if real}}
5      index_state = findVariable("{{name}}", state);
6      if (index_state != -1) {  // -1 means variable not found
7        readInputVariableDouble(index_state,{{fmi.valueReference}});
8      } {{/if}}
9      // ... code for updating other variable types omitted for brevity
10     {{/if}}{{/if}}
11   {{/each}}
12
13   // handle user action
14    handleUserAction();
15
16   // execute a simulation step
17   sendToPVSio("{{tick}}");
18   receiveFromPVSio();
19
20   // update output variables
21   {{#each variables}}{{#if fmi}}{{#if output}}
22     {{#if real}}
23     index_state = findVariable("{{name}}", state);
24     if (index_state != -1){ // -1 means variable not found
25       writeOutputVariableDouble(index_state, {{fmi.valueReference}});
26     } {{/if}}
27     // ... code for updating other variable types omitted for brevity
28     {{/if}}{{/if}}
29   {{/each}}
30 }
```

Listing 1.4: Snippet of the Handlebars template for function `doStep`.

The return of the function is of type `fmi2Status`, which is the standard return type of FMI 2.0 functions invoked by the master – possible return values are `fmi2OK` (the function has been executed correctly) and `fmi2Error` (the function produced an error). The body of the function invokes function `doStep`, which is invoked by the master to trigger the execution of a simulation step, and then returns a constant `fmi2OK` indicating that the step has been executed.

The template for function `doStep` is shown in Listing 1.4. It specifies the four main operations performed by the function: reads input variables of the FMU (lines 3-11); handles user input provided by the graphical front-end by executing the corresponding action in the PVS model and updating the state of the simulation (line 14); executes a step in the PVS model (lines 17-18); receives the new state of the PVS model and updates the output variables of the FMU (lines 20-29). The utility functions used in `doStep` are also specified as Handlebars templates.

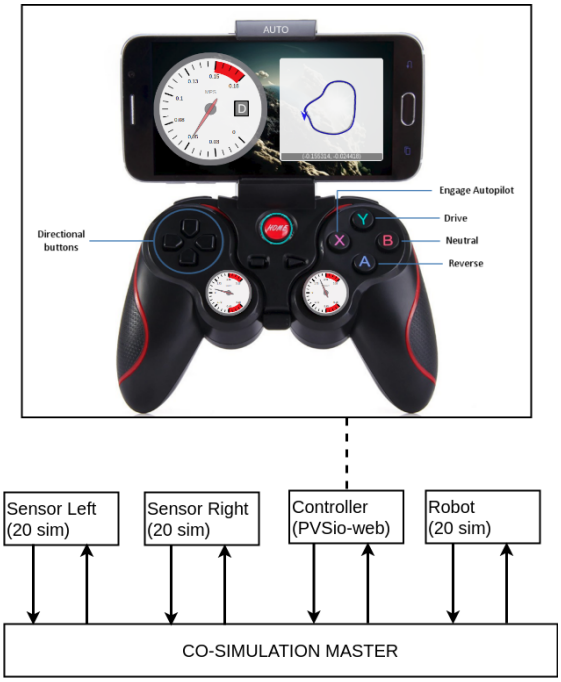

Fig. 2: Co-simulation of line follower robot case study.

4 Demonstrative examples

4.1 Co-Simulation of discrete and continuous components

Our first case study is based on the Line Follower Robot example provided by the INTO-CPS [10] project. In the original example, an autonomous robot has the goal of following a line painted on the ground. The controller of the robot receives the readings from two light sensors placed on the front of the robot (one slightly moved to the left and one slightly moved to the right), and sends commands to the left and right motors which are in charge of the rotation of the left and right wheels, respectively. The INTO-CPS project provides the FMU of the kinematics of the robot (created with the 20-sim [4] tool), the FMU of the sensors (created with 20-sim or OpenModelica [7]), and the FMU of the controller (created with the Overture [9] tool).

In our previous work [20], we replaced the original controller of the robot with a more advanced controller developed with PVSio-web. The new controller allows a driver to override the automatic line following control of the robot, and operate the robot manually, using controls on a dashboard. The sensors and the mechanics of the robot are unaltered with respect to the original INTO-CPS example.

The PVSio-web prototype (shown in Figure 2) provides a navigation display with the trajectory of the robot, two speedometer gauges to monitor the veloc-

(a) U-turn due to high speed. (b) Missed turn.

Fig. 3: Unexpected behaviours of the line follower robot.

ities of the wheels, a speedometer gauge to monitor the velocity of the robot, and various control buttons to allow a driver to accelerate (*up arrow*) or brake (*down arrow*), change direction of the robot (*left and right arrows*), and change gear (buttons *A, Y and B*). There is also a control (button *X*) to switch control mode from manual back to automatic.

In our previous work the PVSio-web prototype was created by manually developing the XML and C code necessary for the FMI interface. In this work we re-created the same prototype automatically, using the APIs of our framework. The new prototype was successfully used in co-simulation scenarios executed using the INTO-CPS Co-simulation Orchestration Engine.

The FMU connected with the PVSio-web navigation display has been used to analyse the robot behaviour when switching control mode from manual to automatic and to expose possible faults of the robot. For example, many experiments pointed out the need to perform a U-turn to get back on track when switching from manual to automatic control and the robot was moving at high speed (see Figure 3a), and some experiments ended up with the robot going far away from the line due to the fact that it reaches perpendicularly the line, decides not to turn and moves on (see Figure 3b).

4.2 Co-Simulation of multiple devices

Our second case study is based on an Integrated Clinical Environment (ICE). In this case, the co-simulation integrates the concurrent execution of three models, each representing a different device (see Figure 4).

ICE is a prototype medical system for intensive care patients. The system includes three devices: a pump infusing morphine; a monitor checking vital signs of the patient; and a supervisor device implementing a safety interlock app that automatically stops the infusion when the patient monitor detects the onset of respiratory depression.

The patient monitor records two vital signs: oxygen saturation level (SpO_2), and Respiration Rate (RRa). The current value of a vital sign is reported using a numeric display. Additionally, a scan-bar trace display shows the temporal evolution of the sign. Each monitored parameter has safe range limits. An alarm is triggered if these limits are exceeded.

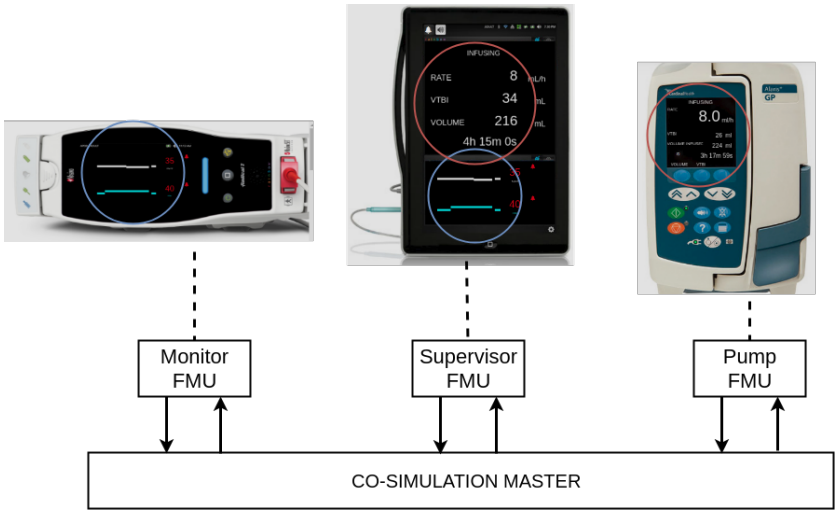

Fig. 4: Co-Simulation of ICE case study.

The front panel of the pump is used to enter the volume to be infused ($VTBI$) and the rate of the infusion of morphine, as well as to start/stop the infusion. During the infusion, the display of the pump shows the rate, the remaining volume of morphine that needs to be infused, and the time to complete the infusion.

The supervisor device has a user interface that can be used for remote monitoring of the pump state and patient monitor state. It is a portable device with a display divided into two sections. The upper section replicates the pump display, and the lower section replicates the patient monitor display.

Starting from the PVSio-web models of these devices, which were already available in the PVSio-web distribution, we used our framework to generate three FMUs, one for each device prototype. These three FMUs were integrated using the INTO-CPS Co-simulation Orchestration Engine, according to the structure shown in Figure 4.

A similar co-simulation example for the ICE system was previously developed in [11] using a (non-standard) co-simulation engine integrated in PVSio-web, which builds on the SAPERE [23] middleware. The migration to the FMI framework did not require any substantial update to the PVSio-web prototypes, as our framework allowed us to re-route and adapt the communication channels used in the SAPERE-based co-simulation to the new FMI interface. The main advantage of the FMI-based co-simulation with respect to that based on SAPERE is that the co-simulation is not limited anymore to PVSio-web prototypes, as other interactive prototypes and system elements developed with tools other than PVSio-web can be integrated in the co-simulation. This is useful, e.g., to introduce patient models in the co-simulation, as tools other than PVS are better suited to specify these models.

5 Related Work

Significant work has been done over the last few years to develop tool support for co-simulation of CPSs. Some works use only one specification formalism for both continuous and discrete systems, like HybridSim [22]. Others support heterogeneous co-simulation [8] with customised solutions, like ForSyDe [21] that supports set of processes that may belong to a distinct Model of Computation, or *OpenICE* [1], that allows the simulation of medical devices for an Integrated Clinical Environment architectures, using a publish-subscribe middleware for communications. In our previous work [2], we developed a CPS co-simulation framework that integrates the Prototype Verification System (PVS) and Simulink.

Recent works use the Functional Mockup Interface (FMI) standard for subsystems synchronisation [19, 20]. In [18] FMI co-simulation is used for modelling and analysing intelligent power systems. Another example is [6], which models the discrete aspects of the system in VDM-RT, the physical part in Modelica and the communication aspects between components in Promela. None of these framework, however, targets modelling and analysis of user interfaces of CPS.

Work on formalising models and proofs for FMI-based co-simulations has been carried out in [24] using Isabelle/UTP and an industrial case study from the railways sector. In [5], a proof-of-concept co-simulation is performed between Ptolemy II and Rodin, using Event-B for formal verification in the aeronautic field. None of these works, however, targets modelling and analysis of user interfaces of CPS.

6 Conclusion and future work

In this paper we present the process for transforming PVSio-web prototypes into FMUs equipped with a standard FMI-2 co-simulation interface. This activity is part of our ongoing work on the development of a framework for formal modelling, simulation and verification of human-centred CPS. In particular, the generation of the FMU, extends our framework making it possible to co-simulate our prototypes with any FMI-compliant co-simulation engine.

Our prototypes can be co-simulated with other prototypes modelled with other tools. For example, in the ICE case, the pump could have been modelled using a different formalism or a model of the patient could be included in the co-simulation. Another advantages of the FMU generation process is that the original PVSio-web prototypes are unchanged, and properties already verified for a prototype are still satisfied by the generated FMU.

Future work will focus on providing a more refined management of the simulated time and a more efficient mechanism for updating the graphical front-end of the prototype. For example, the current implementation has constraints on when time is advanced in the PVSio-web prototype. Specifically, time in the PVS model is advanced only in action *tick* by a discrete step equal to the co-simulation step-size. User actions do not advance time, and they are executed in lockstep with the simulation. The consequence is that only one user action

can be handled at each simulation step. Experience shows that co-simulation steps lower than 250 milliseconds allow for realistic simulations. We plan to remove this constraint by introducing an event-based mechanism for handling user actions continuously over time.

Acknowledgments. Paolo Masci is funded by the ERDF (European Regional Development Fund) through Operational Programme for Competitiveness and Internationalisation COMPETE 2020 Programme, within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency FCT (Fundação para a Ciência e a Tecnologia) as part of project UID/EEA/50014/2013.

References

1. David Arney, Julian M. Goldman, Abhilasha Bhargav-Spantzel, Abhi Basu, Mike Taborn, George Pappas, and Michael Robkin. Simulation of medical device network performance and requirements for an integrated clinical environment. *Biomedical Instrumentation & Technology*, 46(4):308–315, 2012.
2. Cinzia Bernardeschi, Andrea Domenici, and Paolo Masci. A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.
3. Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauß, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauß, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *Proc. of the 9th Intl. Modelica Conference*, pages 173–184. The Modelica Association, 2012.
4. Jan F Broenink. 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5-6):481–492, 1999.
5. Jean-Charles Chaudemar, Vitaly Savicks, Michael Butler, and John Colley. Co-simulation of event-b and ptolemy ii models via fmi. In *ERTS 2014 "Embedded real time software and systems"*, Toulouse, FR, 2014.
6. Luís Diogo Couto, Stylianos Basagiannis, El Hassan Ridouane, Alie El-Din Mady, Miran Hasanagic, and Peter Gorm Larsen. Injecting formal verification in fmi-based co-simulations of cyber-physical systems. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods*, pages 284–299, Cham, 2018. Springer International Publishing.
7. Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. In *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005)*, 2005.
8. Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *CoRR*, abs/1702.00686, 2017.
9. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for vdm. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.
10. Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, et al. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *Modelling, Analysis, and Control of Complex CPS (CPS Data), 2016 2nd International Workshop on*, pages 1–6. IEEE, 2016.

11. P. Masci, P. Mallozzi, FL. DeAngelis, GDM Serugendo, and P. Curzon. Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments. In *Proceedings of the Workshop on Verification and Assurance (Verisure2015), co-located with CAV2015*, 2015.
12. Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. *PVSio-web 2.0: Joining PVS to HCI*, pages 470–478. Springer International Publishing, 2015.
13. Gioacchino Mauro, Harold Thimbleby, Andrea Domenici, and Cinzia Bernardeschi. Extending a user interface prototyping tool with automatic misra c code generation. *arXiv preprint arXiv:1701.08468*, 2017.
14. C. Muñoz. Rapid prototyping in PVS. Technical Report NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA, 2003.
15. CNN News. Tesla in autopilot mode crashes into fire truck, 2018. http://money.cnn.com/2018/01/23/technology/tesla-fire-truck-crash/index.html.
16. CNN News. Uber self-driving car kills pedestrian in first fatal autonomous crash, 2018. http://money.cnn.com/2018/03/19/technology/uber-autonomous-car-fatal-crash/index.html.
17. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in LNCS, pages 411–414. Springer-Verlag, 1996.
18. P. Palensky, A. A. Van Der Meer, C. D. Lopez, A. Joseph, and K. Pan. Cosimulation of intelligent power systems: Fundamentals, software architecture, numerics, and coupling. *IEEE Industrial Electronics Magazine*, 11(1):34–50, March 2017.
19. P. Palensky, A. van der Meer, C. Lopez, A. Joseph, and K. Pan. Applied cosimulation of intelligent power systems: Implementing hybrid simulators for complex power systems. *IEEE Industrial Electronics Magazine*, 11(2):6–21, June 2017.
20. Maurizio Palmieri, Cinzia Bernardeschi, and Paolo Masci. Co-simulation of semi-autonomous systems: The line follower robot case study. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods*, pages 423–437, Cham, 2018. Springer International Publishing.
21. I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, Jan 2004.
22. B. Wang and J. S. Baras. Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems. In *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 33–40, Oct 2013.
23. Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L De Angelis, Giovanna Di Marzo Serugendo, Simon Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, et al. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 17:236–252, 2015.
24. Frank Zeyda, Julien Ouy, Simon Foster, and Ana Cavalcanti. Formalising cosimulation models. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods*, pages 453–468, Cham, 2018. Springer International Publishing.