

Co-simulation of semi-autonomous systems: the Line Follower Robot case study

Maurizio Palmieri^{2,1}, Cinzia Bernardeschi¹, and Paolo Masci³

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, Braga, Portugal

Abstract. Semi-autonomous systems are capable of sensing their environment and perform their tasks autonomously, but they may also be supervised by humans. The shared manual/automatic control makes the dynamics of such systems more complex, and undesirable and hardly predictable behaviours can arise from human-machine interaction. When these systems are used in critical applications, such as autonomous driving or robotic surgery, the identification of conditions that may lead the system to violate safety requirements is of main concern, since people actually entrust their life on them. In this paper, we extend an FMI-based co-simulation framework for cyber-physical systems with the possibility of modelling semi-autonomous robots. Co-simulation can be used to gain more insights on the system under analysis at early stages of system development, and to highlight the impact of human interaction on safety. This approach is applied to the Line Follower Robot case study, available in the INTO-CPS project.

1 Introduction

Cyber-Physical Systems (CPS) are complex physical systems operated by digital controllers. The physical part (the *plant*) may be an entirely engineered system (e.g., a chemical plant) as in traditional control system, but can also be a natural system (e.g., a patient) as in medical applications. From the computational point of view, the existence of digital and physical components requires the use of different kinds of mathematical formalisms, e.g., discrete logic-based models for controllers, and continuous models based on differential equations for plants. In addition, the physical parts of a same CPS may need to be modelled with different languages and tools. Because of this, an efficient way of simulating CPS is by using co-simulation frameworks, which enable integrated simulation of heterogeneous models using multiple tools.

Semi-autonomous systems are a particular kind of CPS. In these systems, the user interface of the system has an important role, as it allows an operator to interact with the system, e.g., to override its autonomous behaviour when desired or necessary. A common example is a car's cruise control that automatically adjusts the speed of the car. The driver can take over control at any time

just by pressing either the brake or the accelerator pedal. Another example is robotic-assisted surgery, where a surgeon console registers the hand’s gestures of a surgeon and translates them into micro-movement of robotic arms.

Simulation and prototyping are important technologies for early detection of design problems in user interfaces of CPS, as they facilitate the discussion of scenarios and design aspects in a multi-disciplinary team of developers, human factors specialists, and domain experts. In critical application domains such as automotive and healthcare, however, simulation and prototyping alone may not be sufficient to ensure that the system meets the safety levels required by regulatory frameworks — they can be used to explore only a finite set of scenarios and user input sequences. Formal verification technologies can be used to extend simulation results and reach the necessary safety level. They are based on mathematical proofs, and allow developers to gain additional confidence that a system meets given safety requirements. It is therefore desirable to integrate as much as possible formal verification with simulation and prototyping frameworks, to make the overall development process more efficient.

In our previous work [2], we developed a CPS co-simulation framework that integrates the Prototype Verification System (PVS) [24] and Simulink⁴. PVS is used for modelling, simulation, and verification of critical CPS software components. Simulink is used for modeling and simulation of continuous aspects of the CPS. Ad-hoc APIs were used to execute two models in lockstep and for time synchronization. In the present work, we enhance this PVS-based co-simulation framework with a Functional Mockup Interface (FMI), a tool-independent co-simulation interface that is becoming a de-facto industry standard.

Contribution. This paper reports on our work on developing an FMI-compliant interface for the Prototype Verification System (PVS) [24] that includes the capability of a GUI and enables human-in-the-loop co-simulation. An example based on a semi-autonomous vehicle is used to demonstrate the utility and capabilities of the developed interface. A controller is modelled and verified in PVS; the PVSio-web [19] toolkit is then used to connect the verified PVS model with a realistic interactive dashboard prototype; a co-simulation is then carried out by using the developed FMI interface to integrate the PVS model and dashboard with the rest of the system components (vehicle’s mechanics, sensors, and environment) simulated with other tools.

Structure. Section 2 presents related work on simulation and verification of CPS. Section 3 illustrates background concepts and tools used in this work. Sections 4 and 5 present the main contributions of this work, i.e., the development of an FMI-compliant interface for PVS, and an example application based on a semi-autonomous system. Section 6 concludes the paper.

2 Related work

In [7], an approach is presented, based on expressing the discrete-event model in the Vienna Development Method (VDM) [8] and the continuous-time model

⁴ <http://www.mathworks.com/products/simulink>

in the Bond-Graph notation [14]. The simulation environment Overture [16] for VDM and the simulation environment 20-sim [5] for Bond-Graphs are integrated into the Crescendo tool [17]. The information needed to co-ordinate the two models, including shared parameters, variables, and events are identified in a contract listing. Synchronization and data exchange is managed by a co-simulation engine.

In [23], the ForSyDe modelling framework [25] is extended to support heterogeneous co-simulation. A ForSyDe model is a hierarchical set of processes, where each process may belong to a Model of Computation (MoC) [18]. A MoC represents the underlying time, synchronization, and communication model assumed by a process. The framework enables processes with different MoCs to co-execute. This framework has been extended with wrapper processes interacting with external simulators or hardware components.

The INTO-CPS project [13] created an integrated tool chain for comprehensive Model-Based Design of CPS based on the Functional Mockup Interface (FMI) standard [3, 4]. The core of INTO-CPS is an FMI-compliant Co-simulation Orchestration Engine that enables a coordinated simulation of heterogeneous models in a distributed environment. Even if many challenges are still open, such as establishing the correctness of the co-simulation of mix continuous and discrete behaviour [6], there are efforts to apply this standard in industry.

Other works address the problem of simulating CPS by using only one specification formalism for both continuous and discrete systems (like for example, HybridSim [27]) or extend original languages to new features, for example, in [26, 12] the integration of MATLAB/Simulink with UML is proposed. For a recent survey of co-simulation technologies for cyber-physical systems, readers may refer to [10].

A complementary approach to the analysis of CPS is formal verification. In this respect, KeYmaera [9] is a theorem prover for differential dynamic logic. It has been applied successfully for proving correctness of automotive, avionics and medical CPS. Our work differs from KeYmaera in that we aim to integrate formal verification by theorem proving in PVS (see for example [1]) with the possibility of performing co-simulation of the system. Our aim is ultimately to facilitate the introduction of formal verification technologies in development processes that routinely use simulation and prototyping.

3 Background

3.1 The Functional Mockup Interface

The Functional Mockup Interface (FMI) [3, 4] is a tool-independent standard to support both model exchange and co-simulation of dynamic models. Co-simulation is performed by a number of *Functional Mockup Units* (FMUs), each responsible for simulating a single sub-model. An FMU contains a sub-model represented in the native formalism of the tool that created it, and the information or tools needed for its execution. That is, an FMU may carry a whole

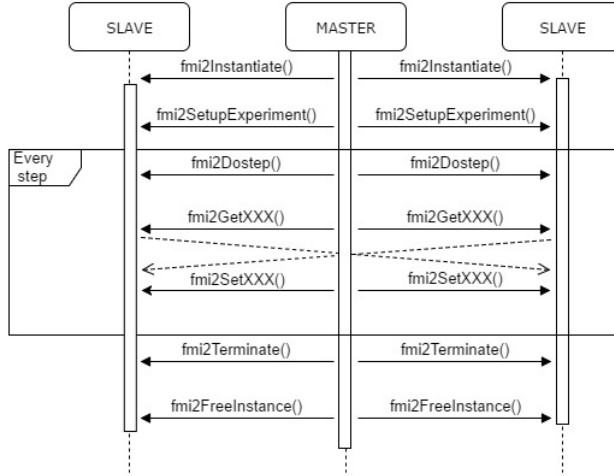


Fig. 1. FMI communication schema.

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 the FMUs acting as *slaves*. The FMI defines a standard API for the FMUs and standard methods of deployment for them.

The FMU’s APIs include functions called at the initialization phase, functions to trigger one simulation step (`fmi2DoStep()`), and functions to exchange data. The latter have a standard signature `fmi2Get<TYPE>` and `fmi2Set<TYPE>`, where `<TYPE>` is a concrete type name, e.g., *Integer* or *Real*. These functions are used to transmit data from and to the FMUs, respectively. Other two functions, `fmi2Termminate` and `fmi2FreeInstance`, can be used to terminate simulation and release resources. Figure 1 shows the communication pattern for these functions.

3.2 INTO-CPS

INTO-CPS [15] is a EU-funded project that is finalizing the development of an integrated tool-chain for model-based design of CPS based on FMI-compliant co-simulation. The tool-chain currently supports simulation of models produced by various tools, including OpenModelica, 20-sim, and Overture. SysML is used to specify the overall architecture of the system to be simulated, by representing the interconnections among the sub-models.

In this work, we embrace the INTO-CPS tool-chain and a case study developed in the INTO-CPS project (the Line Follower Robot⁵), and use them to

⁵ https://github.com/into-cps/case-study_line_follower_robot

demonstrate the FMI extensions we have developed for PVS (additional details on the case study are in Section 5).

3.3 The Prototype Verification System (PVS)

The Prototype Verification System (PVS) [24] is an interactive theorem proving environment for higher-order logic. The PVS specification language provides basic types, such as Booleans, naturals, integers, reals, and others, and type constructors to define more complex data-types (e.g., *records*) and *subtypes*. The mathematical properties of each type are defined axiomatically in a set of fundamental theories, called the *prelude*. New types are declared using the **TYPE** keyword. A *record* type is a tuple whose elements are referred to by their respective *field* name. For example, given the declarations:

```
Wheels: TYPE = [# left: real, right: real #]
axle: Wheels = (# left := 1.0, right := 0.5 #)
```

the expressions `left(axle)` and `right(axle)` denote the speeds of the left and right wheels of `axle`, respectively. Equivalent notations are `axle'left` and `axle'right`. The *override* expression `WITH [..]` can be used for in-line re-definition of record field values. For example, for the declarations above, the expression `axle WITH [left := -1.0]` denotes the record value `(# left := -1.0, right := 0.5 #)`. An example PVS subtype is the following:

```
Speed: TYPE = { x: real | x >= -1 AND x <= 1 }
```

which defines type `Speed` as a subtype of real numbers in the interval $[-1, 1]$. Subtypes can be used in function definitions to define safety constraints and limits that shall be checked. We will use these feature in Section 5, to verify that, e.g., the velocity commanded by the controller does never exceed the robot's mechanical specifications.

3.4 PVSio and PVSio-web

PVSio [22] is a ground evaluator that computes the value of ground (variable-free) expressions. The PVSio evaluator acts as an interactive interpreter for the logic language of PVS, and can be used by developers to simulate a PVS model. At the PVSio prompt, the user types a ground PVS expression (which is equivalent to a function call of imperative languages) and PVSio returns the result of the evaluation. For example, if a PVS theory contains the following function definition

```
compute_velocity(v:real, a:real, t:real): real = v + a*t
```

then its value for a particular triple of arguments can be computed with the following function application: `compute_velocity(3.5, 1.8, 3.0)`;

PVSio-web [19] is an open source toolkit that extends PVSio with functions for creating and executing realistic user interface prototypes based on PVS models. Using PVSio-web, developers can define the visual appearance of the prototype, as well as capture user actions over input widgets, and render feedback on displays and other output widgets on the user interface. These functionalities are used in Section 5 to create an interactive dashboard for driving a Line Follower robot.

4 Development of an FMI-compliant interface for PVS

We developed an FMI-compliant interface for PVS by creating a C wrapper module that implements the FMI interface and spawns a PVSio process. At the beginning of the co-simulation, the wrapper starts a PVSio instance, loads a given PVS model in PVSio, creates a server module to exchange commands with interactive GUI, and waits for input. In more detail, the wrapper performs the following two actions at each simulation step:

- Translates calls to FMI functions into appropriate commands for PVSio;
- Receives replies from PVSio and stores them into appropriate buffers.

Communication between the wrapper and PVSio relies on standard Unix pipes, and communication between the wrapper and GUI uses the WebSocket⁶ protocol. This latter choice allows us to have a loose coupling between the FMU and the GUI, which promotes separation of concerns between the visual appearance of the user interface from its functional behaviour (Model-Controller-View architectural pattern). The overall architecture is summarized in Figure 2.

4.1 Implementation of the FMU

The FMU module implements four core functions: `fmi2Instantiate`, which initializes the FMU; `fmi2DoStep`, which executes a simulation step; and a battery of `fmi2Get/fmi2Set` functions for data exchange. To use a PVS model in the FMU, the model needs to provide at least two functions: `init`, that initializes the PVS model to the initial state (where the state is represented using a PVS record type); and `step`, that performs a simulation step.

The initialization function (`fmi2Instantiate`) starts the PVSio interpreter, redirects the PVSio standard I/O towards a Unix pipe, invokes the `init` function of the PVS model, and stores the result of the evaluation in a variable of the FMU. Finally, a WebSocket server necessary for communication with a PVSio-web prototype is created.

The `fmi2DoStep` function sends PVSio a string encoding a call to the PVS function `step`. A function argument is included, representing the current state of the system. The result of the evaluation is stored in a variable in the FMU.

⁶ <https://www.websocket.org>

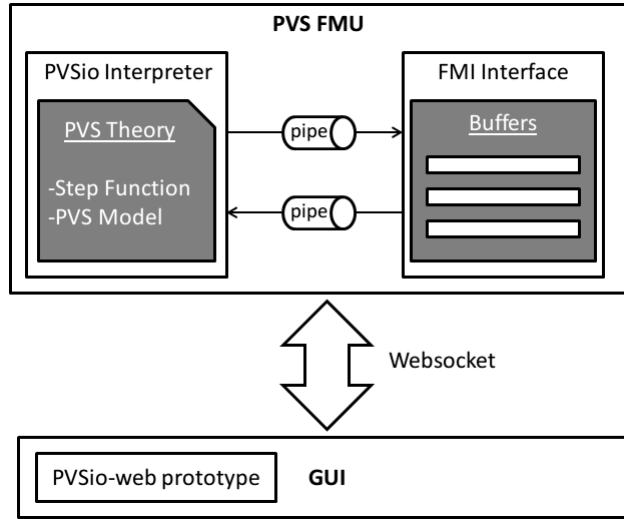


Fig. 2. Architecture of the FMU module for PVS.

4.2 Implementation of the PVSio-web prototype

The PVSio-web prototype is a JavaScript module executed in a web browser. The prototype builds on the PVSio-web communication architecture to open a Websocket connection to Websocket server executed in the PVS FMU. A picture is used as a basis to create the visual appearance of the prototype (e.g., to create the remote dashboard controller prototype illustrated in Section 5, we used a photo of a joypad and a smartphone — see left side of Figure 4). Control and display elements in the prototype are created using a library of input and output widgets provided by PVSio-web. Button widgets capture user actions on certain areas of the prototype (e.g., pressing a button) and translate these actions into commands for the FMU. Display widgets are used for rendering visible state attributes of the system, using an appropriate visualization style (e.g., the velocity of a vehicle can be rendered with speedometer gauges). The FMU, in turn, translates these commands into calls to functions defined in the PVS model. State updates are periodically sent by the FMU to the PVSio-web prototype, to keep the user interface updated with the actual system state.

5 Case study

Our case study is based on the Line Follower Robot example provided by the INTO-CPS European project. In the original example, an autonomous robot (see Figure 3) 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, and sends commands to the left and right motors which are in charge of the rotation of the left and right wheels, respectively. The aim of the controller is

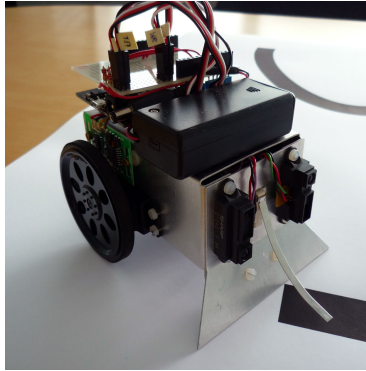


Fig. 3. The INTO-CPS Line Follower Robot (from [13]).

to keep the robot on a path (the dark line in Figure 3). The INTO-CPS project provides the FMU of the robot mechanics (created with the 20-sim tool), the FMU of the sensors (created with 20-sim and OpenModelica), and the FMU of the controller (created with the Overture tool). It also provides a SysML model necessary to link these components.

In the present work, we replaced the original controller of the robot with a more advanced controller developed in PVS. 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 prototype of the dashboard (see left side of Figure 4) provides a navigation display with the trajectory of the robot, two speedometer gauges to monitor the velocities of the wheels, a speedometer gauge to monitor the velocity of the robot, and various control buttons to allow a driver to *accelerate* or *brake*, change direction of the robot (*turn right*, *turn left*), and change gear (*drive*, *reverse*). There is also a command (*home*) to switch control mode from manual back to automatic. Velocity and trajectory shown on the dashboard mirror sensor data communicated to the dashboard through the FMI interface. The original SysML model has been modified to include these new communication links.

In the following sub-section, a description of the PVS controller and the PVSio-web prototype of the dashboard are presented. The full example, including the PVS theory and the PVSio-web prototype, can be downloaded from our repository on github⁷.

5.1 PVS theory of the new controller

Theory `advanced_controller` defines the characteristics and functionalities of the new controller. The initial part of the theory defines the structure of the controller state (lines 9-13), and the data-types of the state attributes:

⁷ <https://github.com/PaoloMasci/pvsio-web-fmi-extensions>


```

1 advanced_controller: THEORY BEGIN
2   %% type definitions
3   LSR: TYPE = { x: nonneg_real | x <= 255 }
4   LightSensors: TYPE = [# left: LSR, right: LSR #]
5   Speed: TYPE = { x: real | x >= -1 AND x <= 1 }
6   MotorSpeed: TYPE = [# left: Speed, right: Speed #]
7   ControlMode: TYPE = { AUTO, MANUAL }
8   %% controller state
9   State: TYPE = [#
10    lightSensors: LightSensors,
11    motorSpeed: MotorSpeed,
12    gear: Gear,
13    cm: ControlMode #]
14   %% ...more definitions omitted for brevity
15 END advanced_controller

```

Field `lightSensors` in the state of the controller (line 10 in the snippet above) holds the input values received from the light sensors, ranging from 0 to 255, according to the robot's sensors specifications; `motorSpeed` holds the rotation speed of the robot wheels, ranging from -1 to 1, according to the robot's mechanical specifications; `gear` is an extension used to represent possible gears of a car-like system. It can be `DRIVE` or `REVERSE`; `cm` stores the control mode, which can be either `AUTO` or `MANUAL`.

The `step` function called at every simulation step updates the rotation speed of the left and right motors when `cm` is in mode `AUTO`.

```

1 step(st: State): State =
2   IF cm(st) = AUTO
3   THEN st WITH [
4     motorSpeed := (#
5       left := update_left_motor_speed(st),
6       right := update_right_motor_speed(st)
7     #) ] ELSE st ENDIF

```

Function `update_left_motor_speed` (`update_right_motor_speed`) in the snippet above updates the left (right) motor speed using a simple control algorithm based on a threshold and the current light sensors reading. The same algorithm was also used in the Overture model of original controller of the robot.

```

1 update_left_motor_speed(st: State): Speed =
2   LET ls = lightSensors(st)
3   IN COND ls`right < 150 AND ls`left < 150 -> 0.4,
4           ls`right > 150 AND ls`left < 150 -> 0.5,
5           ls`right < 150 AND ls`left > 150 -> 0.1,
6           ELSE -> motorSpeed(st)`left ENDCOND

```

In the snippet above, the `LET-IN` construct introduces local definition that can be used in the expressions following `IN`. The `COND-ENDCOND` expression is a many-way switch composed of clauses of the form *condition* \rightarrow *expression*, where all

conditions must be mutually exclusive and must cover all possible combinations of their truth values (an `ELSE` clause provides a catch-all). The PVS type checker verifies that these constraints are satisfied.

For each control provided on the dashboard, the PVS theory provides a matching function. For example, the *accelerate* button is associated with the PVS function `accelerate`, which is defined as follows:

```

1 accelerate(st: State): State = st WITH [
2   cm := MANUAL,
3   motorSpeed := (#
4     left := COND
5       gear(st) = DRIVE
6         -> inc_CW_speed(motorSpeed(st)`left, ACC_STEP),
7       gear(st) = REVERSE
8         -> inc_CCW_speed(motorSpeed(st)`left, ACC_STEP),
9     ENDCOND,
10    right := COND
11      gear(st) = DRIVE
12        -> inc_CCW_speed(motorSpeed(st)`right, ACC_STEP),
13      gear(st) = REVERSE
14        -> inc_CW_speed(motorSpeed(st)`right, ACC_STEP),
15    ENDCOND #)]

```

When function `accelerate` is executed, the control mode is automatically changed to `MANUAL` (line 2 in the snippet above). The speed of the robot is increased by updating the rotation speed of the left and right motors of the robot by an acceleration step `ACC.STEP`. The specific direction of rotation of the motors (clockwise, or counter-clockwise) depends on the *gear* selected by the driver, and on which wheel the motor controls (e.g., to move the robot forward, the left motor needs to rotate clockwise, but the right motor needs to rotate counter-clockwise). When the gear is `DRIVE` (lines 5-6 and 11-12) the rotation speed of the left and right motors is set to move the robot forward. When the gear is `REVERSE` (lines 7-8 and 13-14) the rotation speed of the left and right motors is set to move the robot backwards.

For the developed theory, the PVS type-checker automatically generated 22 proof obligations to ensure correct use of types, coverage of conditions, and disjointness of conditions. All proof obligations were automatically verified by the PVS theorem prover in just 0.29 seconds on a standard laptop (Intel Core i7-5500U, 8GB RAM). This ensures that the developed model does not have bugs such as division by zero, or logic errors such as erroneous control definitions that overshoot the robot's motors specifications.

5.2 PVSio-web prototype of dashboard

We developed a dashboard prototype in PVSio-web (see Figure 4) to allow a driver to interact with the PVS controller and, by doing so, operate the robot and monitor its speed and position. The prototype is based on a joypad: directional arrows are used to steer the robot, and a number of buttons can be used to effect



Fig. 4. Screenshot taken during a co-simulation run. The dashboard prototype is on the left. The output of the INTO-CPS co-simulation window is on the right.

actions of the robot. For example, if the robot had a mechanical arm, some of the buttons could be used to move the arm, or if the robot represents a car-like vehicle, as in our case, they can simulate a gear shift control.

The developed dashboard prototype uses directional arrows to control the direction and accelerate/brake. The *home* button at the centre of the joypad can be used to activate automatic control mode. These interactive controls were created by overlaying the picture of the joypad with transparent interactive areas that can detect button presses. Gears can be shifted using the joypad buttons *Y* (drive), and *A* (reverse). This is the standard approach used in PVSio-web to create interactive controls.

A smartphone mounted at the top of the joypad is used to render speedometer with the current speed of the robot and the current gear, and a navigation display with the current position and direction of the robot on a map. A frameless display at the top of the smartphone shows the control mode (auto/manual). Two additional gauge displays are placed at the bottom of the joypad, to monitor the current rotation speed of the wheels. All these display elements were created by overlaying the picture of the joypad with digital displays available in the PVSio-web distribution. An external JavaScript library ([d3-gauge-plus](https://github.com/gimbo/d3-gauge-plus)⁸) is used to render gauges. The navigator display is implemented using HTML5 Scalable Vector Graphics (SVG). The gauge and navigator displays are part of a new domain-specific library for the PVSio-web, which will be released with the next version of the toolkit.

5.3 Results

Several co-simulation runs were performed by connecting the PVS FMU to the INTO-CPS Co-Simulation Engine. All experiments were configured with a fixed

⁸ <https://github.com/gimbo/d3-gauge-plus>

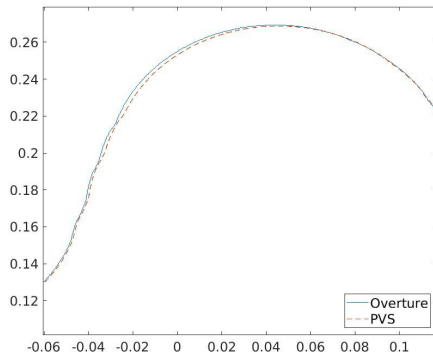


Fig. 5. Control mode validation.

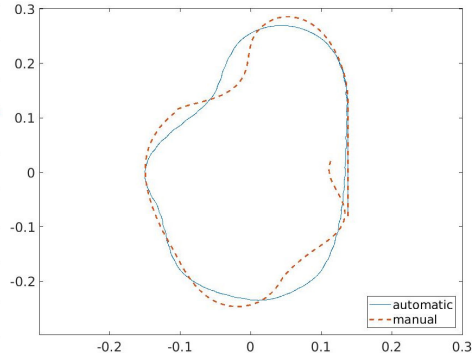


Fig. 6. Automatic vs. Manual drive.

simulation step of 0.01 seconds, and a duration of 35 seconds of simulated time. Figure 4 shows a screen-shot from an ongoing simulation. On the left side, the image shows the joypad prototype displaying the trajectory and speed of the robot. On the right side, the image shows the INTO-CPS application window hosting the co-simulation.

As a first experiment, we checked that the behaviour of the new PVS controller in automatic control mode was the same of that produced by the original Overture controller developed in the INTO-CPS project. As shown in Figure 5, this check was successful: the trajectory of the robot is identical in the two cases (modulo small differences due to mathematical approximation).

Other experiments were then performed to check that the robot was following the commands given by a driver with the dashboard prototype. In one experiment, we tried to use the manual drive to follow the same path as the automatic controller, obtaining the result in Fig. 6 (manual driving is shown with a dashed line). The plot shows that it is possible to approximately follow the same path. The low accuracy in some sections of the track are mainly due to the relatively high speed used by the driver to move the robot, which did not allow an accurate control of the robot's direction.

Finally, experiments were also performed to check the robot behaviour when switching control mode from automatic to manual, and vice-versa. Switching from manual to automatic mode highlighted some interesting scenarios where the robot had an unexpected behaviour. For example, the robot was sometimes overshooting the path so much that a U-turn was then necessary to get back on track (see Figure 7). This happened, e.g., when the control of the robot was initially set to *manual*, the driver accelerated the robot to its maximum speed, and then switched to automatic control mode. When switching from manual to automatic control mode, the robot keeps the same speed set by the driver until a direction adjustment is needed. Because of this, when the robot encounters the line painted on the track, the speed is too high and the directional change issued by the controller is not enough to perform the necessary sharp turn.

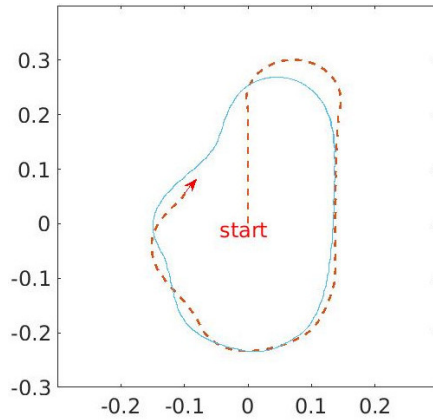


Fig. 7. U-turn due to high speed.

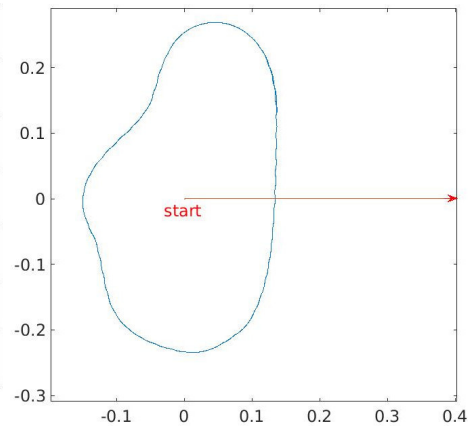


Fig. 8. Missed turn.

Another example abnormal situation is shown in Figure 8, where the robot under automatic control mode does not perform the directional change necessary to bring the robot on track. This seems to be a boundary case of the automatic control algorithm: when the robot reaches the path perpendicularly, both sensors return the same value, and the control algorithm decides not to turn.

6 Conclusions

We have presented the implementation of the FMI-compliant interface for the PVS system. This allows us to use PVS in FMI-based frameworks, such as the INTO-CPS tool-chain. An example based on a semi-autonomous vehicle was developed. The example builds on the Line Follower Robot case study of the INTO-CPS project. We extended the example by developing a new controller that allows an operator to drive manually the robot using a joystick-like controller. The logic of operation of the new controller is entirely specified in PVS, which allows the use of the PVS theorem prover to verify use-related safety properties of the human-machine interface of the system, e.g., consistency of response to user actions, visibility of operating modes, and predictability of response to user commands (see also [11, 20, 21]).

Acknowledgments. We would like to thank the INTO-CPS team for their support with the INTO-CPS tool-chain and the Line Follower Robot example. Paolo Masci's work is financed by the ERDF (European Regional Development Fund) through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme, within the 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 the project UID/EEA/50014/2013.

References

1. Cinzia Bernardeschi and Andrea Domenici. Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System. *Information Processing Letters*, 116(6):409–415, 2016.
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. T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proc. of the 8th Intl. Modelica Conference*, pages 105–114. Linköping University Electronic Press, 2011.
4. 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.
5. J. F. Broenink. Modelling, simulation and analysis with 20-sim. *Journal A*, 38(3):22–25, September 1997.
6. Fabio Cremona, Marten Lohstroh, David Broman, Stavros Tripakis, Edward A. Lee, and Michael Masin. Hybrid co-simulation: It’s about time. Technical Report UCB/EECS-2017-6, University of California, Berkeley, Apr 2017.
7. John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff. *Integrated Formal Methods: 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, chapter Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems, pages 12–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
8. John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. *Vienna Development Method*. John Wiley & Sons, Inc., 2007.
9. F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso. High-assurance spiral: End-to-end guarantees for robot and car control. *IEEE Control Systems*, 37(2):82–103, April 2017.
10. Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *arXiv:1702.00686*, 2017.
11. Michael D Harrison, P. Masci, José C Campos, and Paul Curzon. Verification of User Interface Software: the Example of Use-Related Safety Requirements and Programmable Medical Devices. *IEEE Transactions on Human-Machine Systems*, to appear., 2017.
12. J. Hooman, N. Mulyar, and L. Posta. Coupling Simulink and UML models. In *In Proc. Symposium FORMS/FORMATS, Formal Methods for Automation and Safety in Railway and Automotive Systems*, B. Schnieder and G. Tarnai (eds.), pages 304–311. 2004.
13. INTO-CPS: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems[®], Horizon H2020 project. Grant #644047.
14. Dean Karnopp and Ronald Rosenberg. *Analysis and simulation of multiport systems; the bond graph approach to physical system dynamics*. M.I.T. Press, Cambridge, MA, USA, 1968.

15. P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzon, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6, April 2016.
16. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.
17. Peter Gorm Larsen, Carl Gamble, Kenneth Pierce, Augusto Ribeiro, and Kenneth Lausdahl. *Support for Co-modelling and Co-simulation: The Crescendo Tool*, pages 97–114. Springer, 2014.
18. E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, Dec 1998.
19. 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.
20. Paolo Masci, Rimvydas Rukšenas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon, and Harold Thimbleby. The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering*, 11(2):73–93, 2015.
21. Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. Formal verification of medical device user interfaces using pvs. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2014.
22. C. Muñoz. Rapid prototyping in PVS. Technical Report NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA, 2003.
23. S. H. Attarzadeh Niaki and I. Sander. Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 238–247, June 2011.
24. S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg, 1992.
25. I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, Jan 2004.
26. Carl-Johan Sjöstedt, Martin Törngren, Jianlin Shi, De-Jiu Chen, and Viktor Ahlsten. Mapping simulink to uml in the design of embedded systems:investigating scenarios and transformations. In *OMER4 Post-proceedings, 2008*, pages 137–160, 2008. QC 20100810.
27. B. Wang and J. S. Baras. HybridSim: A Modeling and Co-simulation Toolchain for Cyber-physical Systems. In *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*, pages 33–40, 2013.