UNIVERSITÀ DEGLI STUDI FIRENZE

PhD Program in Smart Computing
Dipartimento di Ingegneria dell'Informazione (DINFO)

Co-simulation and verification of Cyber-Physical Systems using logic models.

Maurizio Palmieri

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Smart Computing

PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Co-simulation and verification of Cyber-Physical Systems using logic models.

Maurizio Palmieri

XXXII ciclo — October 2020

To my family

Acknowledgments

Firstly, I would like to thank my advisors, professors Cinzia Bernardeschi and Giuseppe Anastasi, for the precious help during my whole Ph.D. Their guidance has been crucial for my research and my doctorate activities. I would also like to thank my supervisors, professors Alessandro Fantechi and Gigliola Vaglini, for providing suggestions to my research activities throughout all my Ph.D. I also valued the help supplied by Andrea Domenici and Paolo Masci on the tools used during my Ph.D.

One of the most formative experiences has been the period spent at Aarhus University, where professor Peter Gorm Larsen mentored me by providing useful pieces of advice. I really enjoyed and appreciated all the time he dedicated to my work, and I really thank him for the opportunity. My time in Aarhus was nice also thanks to the many friends I have met: Casper, Hugo, Federica, Camilla, Fotini, Daniel. Thank you so much for the nice time spent together.

I would also like to thank all my office teammates at Pisa University because they helped me enjoy every day of these three years: Michele, Gabriele, Alessandro B., Alessandro R., Carlo, Abdullah, Chiara, Marco, Luca, Antonio, Francesca. You are the main reason why I am very happy to get to work every day. I also want to thank all my friends in Pisa and Monopoli for supporting and encouraging me: Alessandro I., Alessandro T., Nicole, Francesco B., Pierfrancesco, Billa, Marta, Michele, Bledar, Francesco D., Fabio e Daniela.

At last, the biggest thanks go to my family, supporting me since forever.

Abstract

This thesis proposes a methodology to validate and formally verify Cyber-Physical Systems (CPS) exploiting logic for formal models, emerging standard technologies for co-simulation and theorem proving technology for verification. Co-simulation enables the global simulation of a complex system by composing the simulations of its parts, created with different tools without the need to formalize the whole system with a formal language, which, if feasible, is a time-consuming task. The same models of components can successively be used for verifying properties of the whole system.

The proposed methodology also allows to build early prototypes of human-machine interfaces based on formal methods, supporting the work of formal methods experts in charge of analyzing safety-critical aspects of user interfaces in CPS. Formal methods experts usually need to engage with domain experts that may not fully understand the mathematical details of formal analysis. Co-simulation with human-machine interfaces mitigates the barriers mentioned above, providing a speed-up in the design phase of the CPS and improving the validation of formal models.

The framework presented in this thesis extends an existing prototyping toolkit, based on the Prototype Verification System (PVS), with novel functionalities for automatic generation of interactive prototypes supporting the Functional Mock-up Interface (FMI), a de-facto standard technology for co-simulation. The architecture of the framework is presented, along with verification of fundamental aspects of its functionalities. The framework is the core element for the integration of formal models, co-simulation and verification. The PVS theorem prover can be used to verify safety properties of the system under analysis.

Finally, this thesis provides a collection of case studies to show the possibilities offered by the proposed methodology and the developed framework for model-based design of CPS from different application domains.

Contents

Chapter 1

Introduction

1.1 Model-based design and co-simulation

Model-based design is an approach to designing complex systems (Balasubramanian et al., 2006; Jensen et al., 2011). It creates representations (*models*) of a system to assess the characteristics and functionalities of the system throughout its life-cycle, *before* a concrete implementation is built. Model-based simulation technologies applied at the early stages of system design allow developers to gain extra confidence that the system behaves as expected (Franceschini and Macchietto, 2008). In order to produce reliable simulations in the model-based analysis of CPS, developers need to take into account the whole cyber-physical context, which means that software sub-components should be studied together with physical sub-components of the system. Traditionally, the integration is approached by simulating only one sub-component in detail, with the proper language and tool, while simplifying the remaining parts (Palensky et al., 2017a).

The integration of different sub-components, each modeled and simulated with the most appropriate tool, can be achieved with *co-simulation* (Gomes et al., 2018). Co-simulation consists of the theory and techniques to enable global simulation of a coupled system via the composition of different simulators that act as one. Each simulator is considered as a black box capable of exhibiting behavior, consuming inputs, and producing outputs. As a consequence, it is possible to simulate the complete system without any simplification. The simulators are joined by dynamically connecting different models using their inputs and outputs so that the output of one simulator becomes the input of another one. The data exchange, uniform time advancement, and execution coordination are, in the most general case, orchestrated by a master algorithm, which manages the entire co-simulation.

The simulators that compose a co-simulation need to exchange data with each other during all the phases of the co-simulation. Various simulating tools offer several interfaces based on proprietary application program interfaces (APIs), or transmission control protocol (TCP) socket interfaces. One interface type which is gaining consensus as the standard for coupling physical models and simulators is the Functional Mock-up Interface (FMI) (Blochwitz et al., 2012). FMI co-simulation enables a compositional approach to system simulation, where each of the system's components is simulated by a Functional Mock-up Unit (FMU), and a co-simulation orchestrator can be used to connect the FMUs and perform joint simulations.

1.2 Model-based design and formal methods

Formal methods technologies are mathematically-based techniques for systematically checking the properties of a model (Wing, 1990). Developers can rely on formal methods within a model-based design approach to gain the confidence that a system design satisfies given requirements.

Simulation and formal verification are complementary processes, both required in the development of complex systems with safety-critical requirements. Formal methods enable developers to deal with safety issues using well-proven tools of logic and mathematics, providing strong assurance on compliance with requirements. On the other hand, there is the possibility to formalize wrong hypotheses or to prove conclusions derived from wrong specification. It is also possible to produce merely wrong proofs, but the use of automatic model checking (Clarke, 1997) or interactive theorem proving (Owre et al., 1992) mitigates this risk. Simulation provides validation of the model at early stages of development, besides being a prototyping tool supporting the design space exploration. Within a multi-disciplinary team, the typical workflow followed by formal methods experts is as follows (see Figure 1.1):

1. **Formalization of critical system components.** During this phase, the model under analysis is written with a formal language exploiting information available in design documents or obtained by developer experience.

2. **Validation of formal models.** Formal methods experts share the formal model with the rest of the team to check that it correctly represents the intended behavior of the model.

3. **Verification of formal models.** Natural language requirements are translated into mathematical formulae, and formal methods tools are used to verify that the formal model satisfies the formulae.

4. **Validation of formal analysis results.** Results of the formal verification are shared with the other team members, to check whether the results of the verification point out real properties of the model or false positive due to approximations introduced during the formalization process. If false positives are

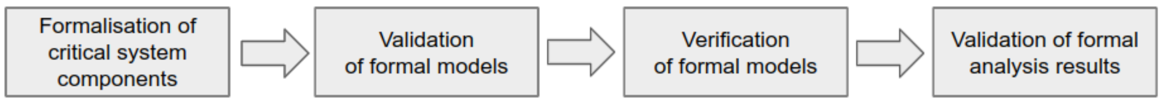

Formalisation of critical system components
Validation of formal models
Verification of formal models
Validation of formal analysis results

Figure 1.1: Workflow typically followed by formal methods experts.

identified, the formalization may need to be revised, and the formal analysis process iterated.

The current generation of formal methods tools provides little support for validation activities. The typical output of formal methods tools is text-based, rich in mathematical details that are not easy to understand: this creates substantial communication barriers between formal methods experts and the rest of the team, and ultimately introduces unnecessary delays in the development life-cycle.

Moreover, outside safety-critical domains, such as automotive or medical devices, the use of formal verification tools is still somewhat limited. One of the reasons is the steep learning curve of verification technologies, which creates an initial cost that discourages the usage.

Model-based design including Graphic User Interface

Complex CPS, like medical devices and avionics systems, are typically developed by a multi-disciplinary team. Consider a medical device such as a dialysis machine, for example. The team would include (Harrison et al., 2019): domain experts, responsible for defining the characteristics and functionalities of the system; software engineers, responsible for developing the software of the final system; clinicians, the operators of the system; formal methods experts, responsible for verifying the the safety claims about the critical component; human-factors specialists, responsible for designing the human-machine interface.

When validating formal models created to analyze human-machine interaction, a simulation of both the human-machine interface and other system components is usually necessary. Interactive prototypes driven by formal models can be used by formal methods experts as a means to discuss formal models and verification results with the rest of the team, without the need to show any textual output produced by the verification tools. The interactive prototypes resemble the visual appearance of the system developed by the team. Simulation examples produced using the prototypes can, therefore, be understood by all team members.

To date, the research community has devoted most of its effort to the improvement of tools for the co-simulation of cyber and physical components of CPS. Less attention has been dedicated to developing tool support to assess the design of the Human-Machine Interface (HMI) using co-simulation technologies, even though human-CPS interaction is often a relevant aspect of the system, e.g., see the accidents involving self-driving cars (CNNNews, 2018b,a), where the design of the car dashboard exceeded the driver's abilities to take over control in case of emergency.

1.3 Contribution

This thesis is concerned with the use of formal methods technologies in the model-based design of Cyber-Physical Systems (CPS). The main contribution consists of the development of a methodology and a framework for model-based design of CPS that allow:

- the construction of executable formal models in the logic language of the Prototype Verification System (PVS), exported as FMU.
- the use of co-simulation technologies to create an integrated simulation with interactive prototypes based on the logical models and connected with other components generated by other modeling tools.
- the application of the interactive theorem prover of PVS on the formal models to build proofs of safety properties.

More specifically, a template for writing executable logic PVS models is described, along with possible extensions to cope with different CPS domains.

The framework allows the automatic creation of a PVS-based FMU that can communicate with a graphic user interface during a co-simulation, allowing the human-in-the-loop co-simulation. Proofs of the correctness of the generated FMUs are provided, as well.

Finally, the methodology and the framework are applied to some case studies: an Unmanned Aerial Vehicles (UAVs) coordination protocol, a semi-autonomous vehicle with a user interface for remote control and an Integrated Clinical Environment (ICE) of medical devices simulated together with a formal model of the patient. The framework can be download from GitHub[1].

This thesis is organized as follows: Chapter 2 reports related works on CPSs, co-simulation, formal methods and HMI; Chapter 3 reports notions on the tools and the formalisms used in this thesis; Chapter 4 describes the proposed methodology for producing executable PVS theories and for creating a PVS-based FMU; Chapter 5 shows details on the implementation of the proposed framework and proofs of its behavior; Chapter 6 contains three case studies where the proposed methodology is applied and Chapter 7 contains the conclusions. Finally, Appendix A provides an example usage of the PVS theorem prover for one of the case studies.

[1] `https://github.com/mapalmieri/pvsio-web`

Chapter 2

Related work

Significant work has been done over the last decade on developing tools for co-simulation of CPS – see (Gomes et al., 2018) for a detailed survey. Many works focus on hardware, software, and physical aspects of the system but not much attention has been dedicated to the analysis of aspects related to human-machine interfaces, even though the impact of potential design issues in the user interface of a system is a well-known problem (Leveson, 2011; Thimbleby, 2010).

Approaches like HybridSim (Wang and Baras, 2013) investigates whether a single specification formalism could be used for modeling both continuous and discrete time components of the systems. Others take the other way around and adopt heterogeneous co-simulation with customized solutions. An example is ForSyDe (Sander and Jantsch, 2004), a tool that is based on the concept of Model of Computation (Goderis et al., 2007). Another example is OpenICE (Arney et al., 2012), a publish-subscribe middleware specifically addressed for medical devices. In (Bernardeschi et al., 2018; Masci et al., 2014b), a co-simulation framework is developed that integrates PVS (Owre et al., 1992) and Simulink. All these works provide ad-hoc solutions for specific tools and cases.

A more general approach to co-simulation is explored in (Palensky et al., 2017b), where the Functional Mockup Interface (FMI) standard (Blochwitz et al., 2012) is adopted for synchronization of different subsystem models. Similarly, in (Palensky et al., 2017a), FMI co-simulation is used for the analysis of intelligent power systems, and in (Couto et al., 2018), for an air-conditioning system. Works on formalizing models and proofs for FMI-based co-simulation has been carried out in (Zeyda et al., 2018) using Isabelle/UTP in a case study from the railways' domain. In (Chaudemar et al., 2014), a proof-of-concept co-simulation is performed between Ptolemy II and Rodin, using Event-B for formal verification in the aeronautic field.

Additional related work includes the usage of formal methods for the analysis of CPS. In this respect, KeYmaera X (Fulton et al., 2015) is a theorem prover for differential dynamic logic. It has been applied successfully for analyzing automotive, avionics and medical CPS. PVS (Owre et al., 1992) has been used in many domains: (Muñoz et al., 2015) to formally specify and verify a detect-and-avoid algorithm intended to support the integration of Unmanned Aircraft Systems into civil airspace, (Bernardeschi and Domenici, 2016) to verify the property of a water tank control system, (Bernardeschi et al., 2018) to verify the determinism of the control algorithm of a pacemaker, and (Son and Seong, 2003) to verify a safety-critical software for a nuclear power plant protection system. A different branch of formal analysis techniques is based on model checking (Clarke, 1997). Hybrid model checking, which relies on the formalism of Hybrid Automata (Henzinger, 1996), is used for the analysis of CPS. Many hybrid model checking tools have been developed in the literature: HYTECH (Henzinger et al., 1997) which is used to analyze linear hybrid automata, d/dt (Asarin et al., 2002) which is suited to analyze linear continuous dynamics with uncertainties in the inputs, HYCOMP (Cimatti et al., 2015) which is based on Satisfiability Modulo Theories (De Moura and Bjørner, 2011), and SPACEEX (Frehse et al., 2011) which has been used to successfully verify a helicopter control algorithm. These are just few examples of model checking tools.

Many tools for model-based design of user interfaces have been recently developed. SCR (Heitmeyer et al., 1998), for example, supports prototyping and analysis of interactive systems. MARIA (Paterno et al., 2009) and EOFM (Bolton et al., 2011) are specialized for modeling user tasks, and IVY (Campos and Harrison, 2009) is dedicated to formal verification of usability properties, but they all lack features for developing prototypes of graphic user interfaces. Work on co-simulation applied to human-in-the-loop experiments has been carried out in (Pedersen et al., 2017). An example is developed that integrates an interactive simulation of a user interface with hardware and software components. Their objective is to assess the importance of integrating human-machine interface design in the development process. In (Nagele and Hooman, 2017), a co-simulation framework is presented that builds on the High-Level Architecture (HLA) and the FMI interface. HLA is a standard for distributed simulations. The main focus of their work is to explore the integration of HLA and FMI-based simulations. Human-in-the-loop capabilities are explored to a certain extent in the presented case study, where a simple user interface is developed for operating and monitoring a thermostat. The automatic generation of interactive prototypes is out of scope. Others have also used HLA as a co-simulation engine for FMI-based co-simulations. As noted in (Garro and Falcone, 2015), however, using HLA brings several challenges, as the HLA standard uses different assumptions on the nature of the mechanisms for time synchronization and data exchange. The framework developed in (Masci et al., 2015a) uses a middleware (SAPERE (Zambonelli et al., 2015)) as co-simulation engine for connecting many PVSio-web prototypes.

The methodology proposed in this thesis differs from these related works in that it aims to supports co-simulation of HMI interactive prototypes generated by logic specifications. Moreover, formal proofs conducted by the PVS theorem prover can be validated against the results of the co-simulation. The framework described in this thesis allows integration of PVSio-web prototypes with other FMI-compliant components developed with other tools. For example, as it will be shown in Chapter 6, PVSio-web prototypes can be joint with OpenModelica models capturing the physical aspects of the system.

Chapter 3

Background

This chapter provides background notions on cyber-physical systems and on how standard co-simulation can be used to facilitate their design, as well as details on the logic formalism used in this thesis: the formal specification language of the Prototype Verification System, with the associated theorem prover and prototyping environment PVSio-web.

3.1 Cyber-Physical Systems

Cyber-Physical Systems (CPS) are systems integrating computation with physical processes. Embedded components handle the software part of the physical processes, usually with feedback loops where physical variables affect software evaluation and vice versa. CPSs are inherently complex due to the necessary combination of the cyber and physical worlds, which have different time management. The engineering of reliable CPSs requires compositional modeling and analysis techniques that deal with requirements belonging to different domains (e.g., physics, software, chemistry, biology). Within each of these domains, there are many different modeling issues, some of them shared among multiple domains. For example, software models can be synchronous or asynchronous; the physical world is composed of co-existing physical dynamics in a time continuum; and the system may have to conform to several different regulatory constraints. Solving these diverse concerns, ensuring interoperability and providing reliable communication among these components is one of the most relevant scientific problem as it involves different communities . There are considerable challenges, mainly because the physical components of such systems introduce safety and reliability requirements qualitatively different from those in the software ones.

As an example, the autonomous vehicle shown in Figure 3.1 is a simple CPS: a single-axle vehicle, which moves at a constant speed and whose turning speed can be controlled. The controller must be able to steer the vehicle until it reaches

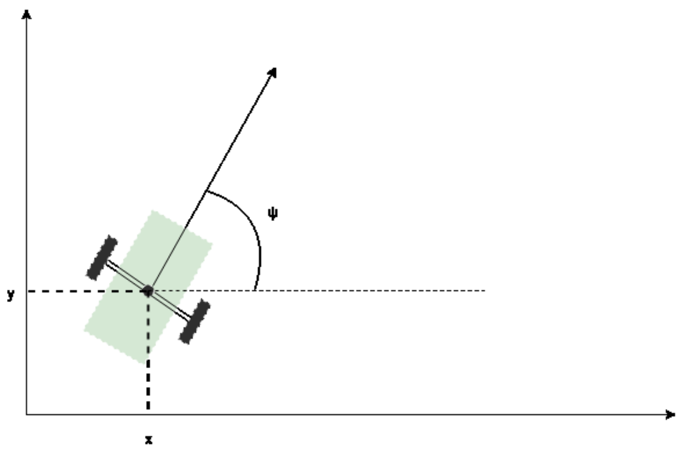


Figure 3.1: Example of a simple autonomous vehicle.

its assigned target. Model-based design of CPS allows us to analyze the system behavior before a physical prototype of the system is built. Simulation is one of the techniques that are usually applied together with testing in the analysis of systems behaviors: an abstract model of a CPS, expressed in some modeling language such as Simulink or Modelica is executed. Figure 3.2 shows a Simulink blackbox model of the vehicle's kinematics and controller.

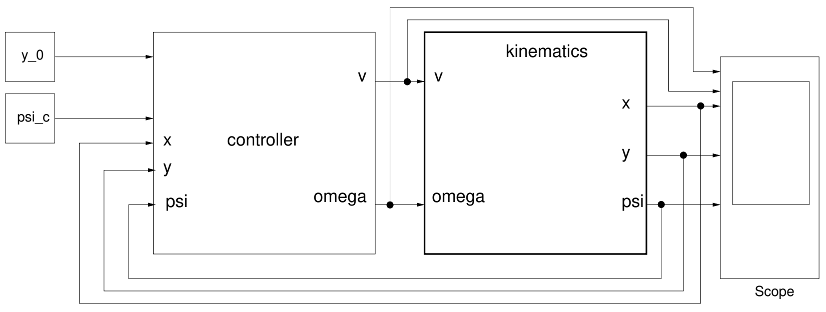
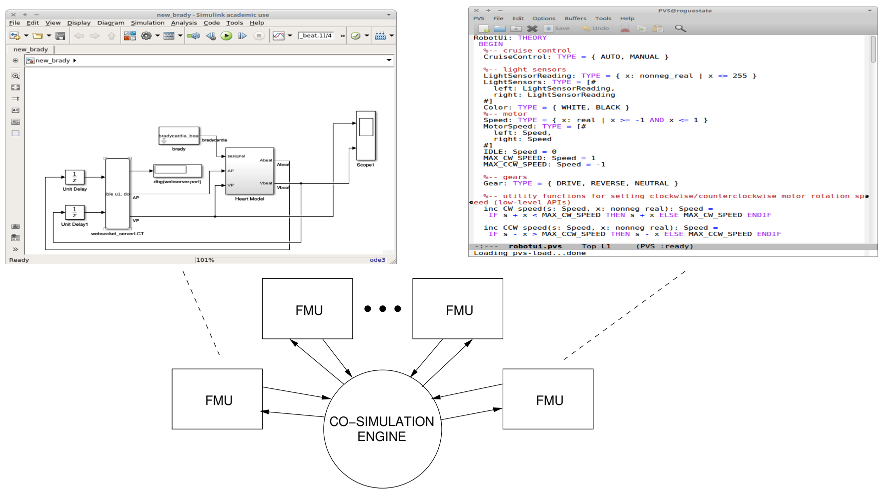


Figure 3.2: Example of a simple CPS.

Applications of CPS have a huge impact on the most recently developed technologies. They include integrated clinical environments, assisted living, traffic regulation, autonomous automotive systems, data mining from sensor networks, avionics, critical infrastructure control (nuclear plants, water resources, and communications systems for example), distributed robotics (telepresence and telemedicine), defense systems, advanced manufacturing, and smart cities.

3.2 Co-simulation with standard interface

CPS specification and simulation must deal with the interactions among the continuous- and discrete-time components that are better modeled with different



```
RobotUi: THEORY
 BEGIN
  %-- cruise control
  CruiseControl: TYPE = { AUTO, MANUAL }

  %-- light sensors
  LightSensorReading: TYPE = { x: nonneg_real | x <= 255 }
  LightSensors: TYPE = [#
    left: LightSensorReading,
    right: LightSensorReading
  #]
  Color: TYPE = { WHITE, BLACK }
  %-- motor
  Speed: TYPE = { x: real | x >= -1 AND x <= 1 }
  MotorSpeed: TYPE = [#
    left: Speed,
    right: Speed
  #]
  IDLE: Speed = 0
  MAX_CW_SPEED: Speed = 1
  MAX_CCW_SPEED: Speed = -1

  %-- gears
  Gear: TYPE = { DRIVE, REVERSE, NEUTRAL }

  %-- utility functions for setting clockwise/counterclockwise motor rotation sp
eed (low-level APIs)
  inc_CW_speed(s: Speed, x: nonneg_real): Speed =
   IF s + x < MAX_CW_SPEED THEN s + x ELSE MAX_CW_SPEED ENDIF

  inc_CCW_speed(s: Speed, x: nonneg_real): Speed =
   IF s - x > MAX_CCW_SPEED THEN s - x ELSE MAX_CCW_SPEED ENDIF
-:---  robotui.pvs    Top L1     (PVS :ready)
Loading pvs-load...done
```

Figure 3.3: FMI co-simulation structure.

mathematical formalisms, and require different simulation tools. For these reasons, co-simulation is a promising approach, as it consists of the coordinated and distributed execution of different simulators.

Co-simulation can be used to study the behavior of the complete CPS. The Functional Mockup Interface (FMI) (Blochwitz et al., 2012) is a tool-independent standard for the co-simulation of dynamic systems. The main elements of an FMI compliant co-simulation are the *Functional Mockup Units* (FMUs), each responsible for simulating a single model in the particular formalism and execution environment used to create the model. An FMU may carry a whole simulation environment (tool wrapper FMU), or just information needed by an FMI-compliant host environment to simulate the model contained in the FMU (standalone FMU). The FMUs are orchestrated by a master algorithm, in charge of exchanging consistent data among the FMUs. An example of FMI co-simulation is shown in Figure 3.3 where two FMUs are designed with different tools (Simulink and PVS).

An FMI-compliant host environment provides a *master* algorithm that orchestrates the execution of other FMUs acting as *slaves*. Figure 3.4 shows the typical execution pattern of an FMI-based co-simulation. Orchestration is obtained through a set of standard APIs, which include: initialization functions; functions for data exchange, such as getters and setters; and a function `fmi2DoStep` that triggers the execution of one simulation step. Getters and setters are in the form `fmi2Get`<*TYPE*> and `fmi2Set`<*TYPE*>, where <*TYPE*> is a concrete type name, e.g., *Integer* or *Real*. Functions `fmi2Get` and `fmi2Set` are executed at the end of each co-simulation step, as soon as `fmi2Dostep` completes.

The interest of industry in the FMI standard has increased steadily in recent years. Many tools support FMI, including Simulink, OpenModelica, PtolemyII, CATIA, and IBM Rational Rhapsody. In the automotive field, MODELISAR (Abel

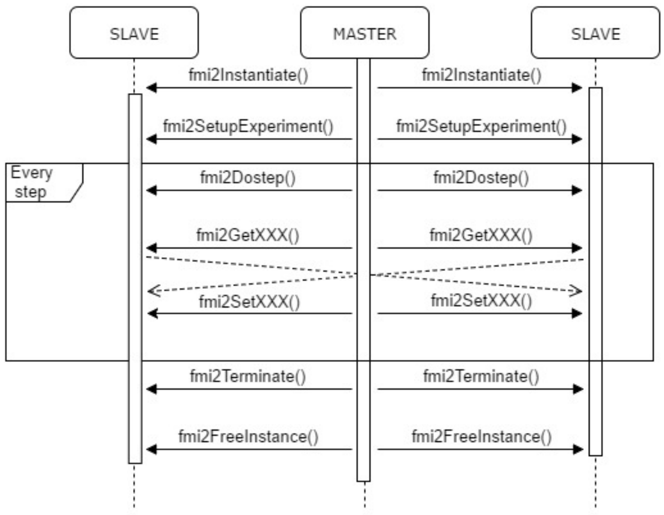


Figure 3.4: FMI execution pattern (master-side).

et al., 2012) uses FMI for model exchanges and supports the AUTOSAR standard; ACOSAR (Krammer et al., 2016) (Advanced Co-simulation Open System Architecture) intends to use FMI for developing Advanced Co-simulation Interface (ACI) for RT-System integration.

INTO-CPS (Larsen et al., 2016) is an example host environment that supports the FMI standard. Case studies in the field of railways, automotive, buildings, and agriculture demonstrate the benefits of the integration of tools and co-simulation technology in industrial settings. Using co-simulation, developers can create prototypes suitable to validate hypotheses embedded in the models and analyze the global behavior in the early stage of design, thus reducing development time, design errors, and costs (Thule et al., 2019).

3.3 The PVS Environment

The *Prototype Verification System* (PVS) (Owre et al., 1992) is an interactive theorem-proving environment whose users can define theories in a higher-order logic language and prove theorems with respect to them.

The PVS specification language provides basic types, such as booleans, naturals, integers, reals, and others, and type constructors to define more complex types. The mathematical properties of each type are defined axiomatically in a set of fundamental theories, called the *prelude*. Among the complex types, the ones used in this thesis are *record* types and *predicate subtypes*.

A *record* is a tuple whose elements are referred to by their respective *field* name. For example, given the declarations:

```
wheels: TYPE = [#
  left: Speed,
  right: Speed #]
axle: wheels =
      (# left := 1.0, right := 2.0 #)
```

`axle` is an instance of type `wheels` and 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 *overriding* operator `:=` in a `WITH` expression redefines record fields. With the declarations above, the expression

```
axle WITH [ left := -1.0 ]
```

denotes the record value `(#-1.0, 2.0#)`.

An example of predicate subtype is the following:

```
LightSensorReading: TYPE =
         { x: nonneg_real | x <= 255 }
```

which represents the real numbers in the $[0, 255]$ interval.

Function declarations are in the form

```
foo(x: T1): T2
```

where foo is the function name, x is a function argument, of type `T1`, and `T2` is the function return type.

The PVS syntax includes the well-known logical connectives and quantifiers, besides some constructs similar to the conditional statements of imperative languages. These constructs are the `IF ... ENDIF` expression and the `COND ... ENDCOND` expression. The latter is a many-way switch composed of clauses of the form *condition* $\rightarrow$ *expression* where all conditions must be mutually exclusive and cover all possible combinations of their truth values (an `ELSE` clause provides a catch-all). The PVS type checker ensures that these constraints are satisfied.

Definitions within a given theory may refer to definitions from other theories. This makes it possible to build complex system specifications in a modular and incremental way.

The PVS environment includes the NASALIB theory libraries (Dutertre, 1996) providing axioms and theorems addressing many topics in mathematics, including real number analysis, and it can be applied to model both the discrete and the continuous part of the system.

The PVS theorem prover is based on the sequent calculus (Owre et al., 1995). The structure of a *sequent* is in the following form, where the turnstile symbol ‘`|--`’ separates the *antecedent* formulae above it from the *consequents* below.

```
{-1} A1
  ...
{-n} An
|-------
{1} B1
  ...
{m} Bm
```

A sequent is proved if (i) any consequent B_i is true, or (ii) any antecedent A_i is false, or (iii) any formula occurs both as an antecedent and as a consequent. The proof of a sequent consists of applying various inference rules until one of the above sequent forms is obtained. A formula to be proved is represented as a sequent without antecedents.

The language of PVS is purely declarative, but its PVSio extension (Muñoz, 2003) can translate PVS function definitions into Lisp code so that a PVS expression denoting a function application with fully instantiated arguments can be interpreted as an imperative function call. The PVSio extension includes input/output functions allowing the system prototype to interact with the user and the computing environment. Moreover, MISRA C code can be automatically generated from PVS theories for automata (Masci et al., 2014b; Mauro et al., 2017), using the PVSio-web tool-set (Oladimeji et al., 2013).

3.4 PVSio-web

PVSio-web (Masci et al., 2015b) is a toolkit for prototyping and analysis of interactive (human-machine) systems. Using the toolkit, developers can create interactive simulations that closely resemble the visual appearance of a real system. An example prototype of a medical device is shown in Figure 3.5.

PVSio-web prototypes consist of two parts: a back-end that executes a formal model defining the behavior of the prototype, a front-end that defines the visual appearance of the prototype.

Formal back-end. The back-end of a PVSio-web prototype defines the behavior of the system, including how the prototype reacts to user actions and other system events. PVSio-web provides a graphical editor for defining such behavior in a user-friendly way: the Emuchart editor. Emucharts are state-machine diagrams containing the following elements:

- a collection of different operational `modes` in which the system operates,
- a collection of all the possible `transitions` among the modes,
- a collection of all the `variables` of the system.

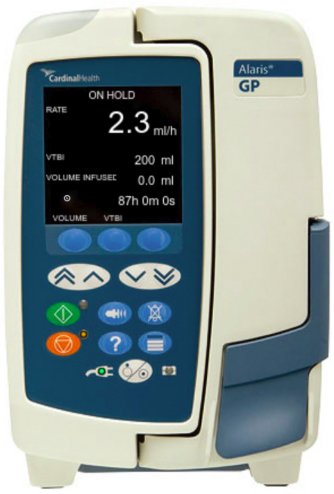


Figure 3.5: Example prototype created with PVSio-web.

This diagram can be automatically translated into different formalisms, including PVS, MISRA C, Java, VDM-SL, and many more. Ultimately, the back-end uses the PVSio (Muñoz, 2003) evaluation environment to compute the evolution of the prototype, exploiting the automatic PVS generation from Emuchart.

The evolution of a PVSio-web prototype is specified using transition functions, i.e., functions that accept one argument, representing the current state of the prototype, and return a new state of the prototype. The state of the prototype is modeled as a record type, where each record represents a variable of the prototype. The state is enriched with two new records that are used to store the current and the previous modes of the prototype.

Graphical front-end. The front-end is responsible for rendering the visual appearance of the prototype and handling events associated with user actions. If available, a picture of the real system is typically used as a basis to create the visual appearance of the prototype, and a set of widgets provided by PVSio-web allows developers to transform the static picture into an interactive simulation. The front-end is executed in a Web browser. Web technologies (HTML5 & JavaScript) are used to create hot-spot areas over the picture and link these areas to functions defined in the back-end. Input widgets are linked with functions representing user actions over user interface elements (e.g., buttons, sliders). The state attributes that are visible on the user interface of the system are rendered in the displays of the prototype, to reproduce the real system. Example prototypes developed with PVSio-web can be found in (Masci et al., 2014a; Bernardeschi et al., 2019) and in training material developed for end users[1].

[1]`https://www.youtube.com/watch?v=T0QmUe0bwL8`

Chapter 4

Co-simulation and Verification with Logic-based models

This thesis proposes an approach to include a PVS executable theory into a co-simulation. The PVS-based FMU is a relevant and characterizing aspect of the proposed approach, as it makes it possible to use the same logic specification for simulation and formal verification. The proposed approach is divided into two processes: on one side, the PVS theory is used to verify properties of the modeled component, and on the other side, co-simulation is used to validate the same theory against other components of the system. Figure 4.1 explains the approach: if both processes end up with success, the output is a verified and validated PVS theory. If one of the two processes fails, then the PVS theory needs improvement, and both processes produce materials that can be used to guide the refinement process. When the improvement is complete, since it has modified the theory, both processes must be executed again. Finally the theorem prover can be used to prove properties of the whole system by building abstract models in PVS for the components described with other languages. These abstract models must retain enough information to guarantee the correctness of the properties under analysis.

4.1 Logic Based models

An executable PVS theory must be specified according to the schematic example shown in Listing 4.1, where:

- `robot_th` is the name of the theory.
- `State` is a record field with all the variables (inputs, outputs, locals, and parameters) representing the state of the component.
- `init_state` is the initial state.

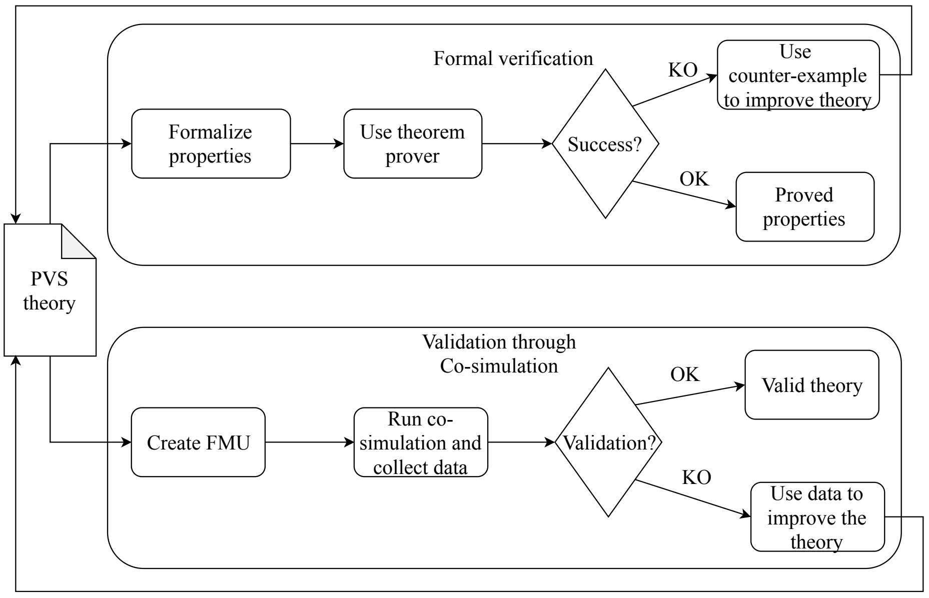

Formal verification
Formalize properties
Use theorem prover
Success?
KO
Use counter-example to improve theory
OK
Proved properties
PVS theory
Validation through Co-simulation
Create FMU
Run co-simulation and collect data
Validation?
OK
Valid theory
KO
Use data to improve the theory

Figure 4.1: Proposed approach for the PVS-based process.

- `tick` is the transition function that takes as argument the current state and returns a new state where the outputs and local variables are possibly changed. In the new state the input variables are unchanged.

Input and output variables are used for communications with other components (in our specific case other FMUs). Generally, in order to take into account the simulation time, the state of the theory is enriched with a variable `time` that is updated by the `tick` function. The skeleton of the theory is intended to model a simple robotic vehicle. This theory can be easily extended to include additional data and functionalities to cope with different applications.

Extension for semi-autonomous control

If the theory above describes a vehicle with autonomous control, such theory can be extended to model also a manual control mode of the vehicle, obtaining a mixed-mode control. In the mixed-mode control theory, `State` is composed of the original variables of the autonomous control, optional variables needed for manual control, and a variable that stores the current control mode. A skeleton of this extension is shown in Listing 4.2 where function `tick` checks `control_mode` to determine the current control mode. When the current mode is `MANUAL` the `tick` function does not modify the state, as other case-specific functions will change the state according to the action performed by the user (e.g., a function for accelerating the robot). An example of manual control function is shown in Section 6.1.

```
robot_th: THEORY
BEGIN

% state of the component
  State: TYPE [#
               %inputs...
               %outputs...
               %local variables...
               %parameters...
               %time... #]

% initial state of the component
 init_state: State = (#
                       %inputs...
                       %outputs...
                       %local variables...
                       %parameters...
                       %time... #)

% transition function
  tick(st: State): State =
        st WITH [
              %local variables...
              %outputs...
              %time... ]
END robot_th
```

Listing 4.1: Schema of an executable theory.

```
semi_autonomous_robot_th: THEORY
BEGIN
ControlMode : TYPE = {AUTO, MANUAL}
State: TYPE [#
               % variables of the original state
               % optional variables for manual control

               control_mode : ControlMode
               #]

%case-specific functions for manual control
 accelerate(st : State) : State = % omitted

 tick(st: State): State =
       IF st`control_mode = AUTO THEN
           % original automatic algorithm
       ELSE
           st WITH [
           % only the time advancement
           ]
END semi_autonomous_robot_th
```

Listing 4.2: Schema of semi-autonomous control.

```
fault_analysis_th: THEORY
  BEGIN
  IMPORTING robot_th

  % state of the component
    ext_State: TYPE [#
                 original_state : State,
                 stepcounter : int,
                 clockS : int,
                 clockA : int #]


  % sensor_fault and actuator_fault implement the failing behaviors
  sensor_fault(st: ext_State): ext_State = ... % omitted
  actuator_fault(st: ext_State): ext_State = ... % omitted

% transition function
  ext_tick(st: ext_State): ext_State =
            LET st1 = sensor_fault(st),
                st2 = st1 WITH (
                      original_state := tick(st1`original_state),
                      stepcounter := stepcounter+1 )
             IN actuator_fault(st2)
  END fault_analysis_th
```

Listing 4.3: Schema of fault analysis extension.

Extension for fault analysis

CPSs are often safety-critical systems, and the analysis of the behavior of the system in case of faults to sensors and actuators is of main concern. The template provided in this thesis can be extended in a modular way by (i) defining a function that describes the effects of the fault on the component state and (ii) by adding variables to the state for recording the number of current co-simulation step and for modeling the step at which the fault occurs. In particular, in case of transient faults, these variables are similar to clocks that can be reset or increased at each co-simulation step. Since faults to sensors affect the inputs to the controller and faults to actuators affect its outputs, the extended version of the tick function first passes the current state to the function modeling sensor faults, and then the resulting state is passed to the original tick function, which computes another state that is further transformed by the function modeling actuator faults, as shown in Listing 4.3.

In the theory shown in Listing 4.3, `sensor_fault` and `actuator_fault` functions model the effect of faults, `stepcounter` stores the co-simulation step and `clockS` and `clockA` identify the step at which the sensor or the actuator becomes permanently faulty.These variables are tested in the `sensor_fault` and `actuator_fault` functions, respectively. According to the type of analysis carried on, one of the two functions `sensor_fault` and `actuator_fault` can be omitted. This kind of extension will be further investigated in the case study of the Line Follower Robot with different policies of fault occurrences.

```
coordination_th: THEORY
 BEGIN
 % the state of the coordination component contains timesteps and
   coordination_rate
 % the coordination_algorithm function implements the specific algorithm

 tick(st: State): State =
           COND
           st‘timesteps >= st‘coordination_rate ->
                st WITH [
                    outputs := coordination_algorithm(st),
                    timesteps := 0
                    ]
           st‘timesteps <  st‘coordination_rate ->
                st WITH [
                    timesteps := st‘timesteps+1
                    ]
 END coordination_th
```

Listing 4.4: Tick function for coordination algorithms.

Modeling multi-agent coordination

The same schema for the executable PVS theory in Listing 4.1 can be adapted to model multi-agent coordination algorithms. When modeling cooperative CPS the coordination protocol, one for each agent, can be modeled as a PVS theory that (i) interacts with the agent component possibly modeled with another tool and (ii) interacts with the coordination theory of other agent.

Figure 4.2 shows an example in which multiple agents cooperate to accomplish their task. Generally, the exchange of information for coordination has a slower rate than the data exchange between the coordination part and the controller part of the single agent. To implement this different rate, a possible skeleton of the PVS transition function `tick` is shown in Listing 4.4, where `coordination_rate` is the number of steps between two synchronizations of coordination components and `timesteps` counts the number of steps since the last synchronization.

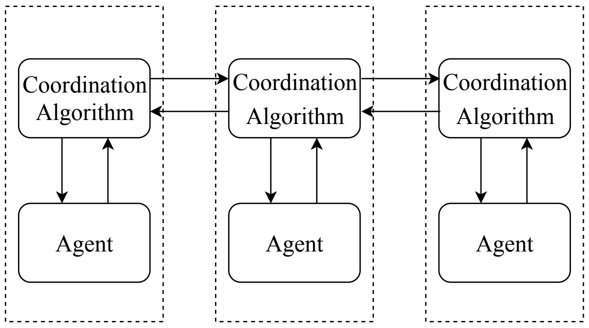


Figure 4.2: Example of general coordination approach.

This kind of extension will be further investigated in the case study of the swarm of UAVs.

4.2 Architecture of a PVS-based FMU

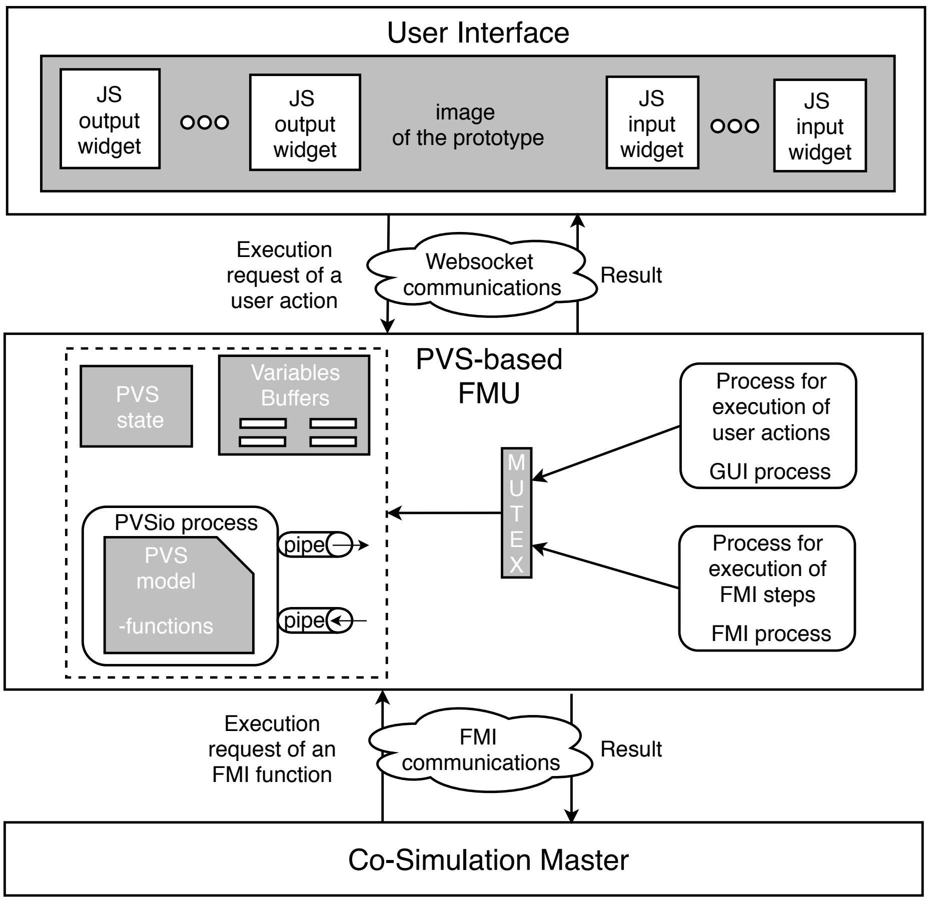


Figure 4.3: Structure of a single FMU.

A diagram representing the structure of a single FMU is shown in Figure 4.3. This structure also includes elements for the management of user action that will be explained in the following. Communication between the co-simulation master and the FMU builds on the standard APIs defined in the FMI standard. Communication between the FMU and the PVSio process is carried out using standard Unix pipes.

The FMU performs a co-simulation step when the master algorithm invokes `fmi2DoStep`. A state variable `PVS state` is used in the FMU to store the state of the PVS model. This is necessary because the native PVSio environment is state-less — it only provides an interactive command line for evaluating PVS expressions. The state of the FMU and the PVSio execution environment represent shared resources for `FMI process` and `GUI process`. A locking mechanism is used to guarantee mutual exclusion. Exclusive locks of the `pthread` standard library are used. A `watchdog` mechanism is implemented to guarantee that locks are released within a given time.

Following the FMI standard, variables of the FMUs are stored in buffers. The FMU buffers are accessed by the master algorithm at the end of each co-simulation step, using `fmi2Get()` and `fmi2Set()`. Information in the `PVS state` is used to update the FMU buffers at each co-simulation step.

The master algorithm periodically calls functions of the FMU to advance the co-simulation. The communication pattern for exchanging commands and data between the FMU and the master algorithm is as follows:

1. The master algorithm calls `fmi2DoStep()` to trigger the execution of a co-simulation step in the FMU;
2. `fmi2DoStep()` waits for permission to access the shared elements of the FMU in mutual exclusion;
3. `fmi2DoStep()` copies the values of the input variables from the buffers to the `PVS state`;
4. Using the pipes connected to PVSio, function `fmi2DoStep()` evaluates the time-advancing function defined in the PVS model. The current value of the `PVS state` is passed as argument to the function. The value returned by the PVSio process is used to update the `PVS state`;
5. `fmi2DoStep()` copies the values of the output variables from the `PVS state` to the buffers;
6. `fmi2DoStep()` releases the permission to access shared elements of the FMU and terminates;
7. The master algorithm calls `fmi2Get()` and `fmi2Set()` in sequence, to acquire the new values of the output variables and update the value of input variables in the FMUs.

Co-Simulation including Graphic User Interfaces

A diagram representing the overall co-simulation architecture is shown in Figure 4.4. The FMUs of PVSio-web prototypes encapsulate the back-end modules defining the behavior of the prototypes. Modules responsible for the visual appearance of the prototypes are external to the FMUs; they communicate only with the FMU and do not interact directly with the co-simulation engine. This design choice promotes a modular architecture and enables hot-swapping of different look and feel of the prototype without restarting the co-simulation. This is useful during system development, e.g., when exploring different design alternatives for the human-machine interface of the prototype.

User actions are performed when the user interacts with the front-end of the prototype. User actions and co-simulation steps are handled with concurrent pro-

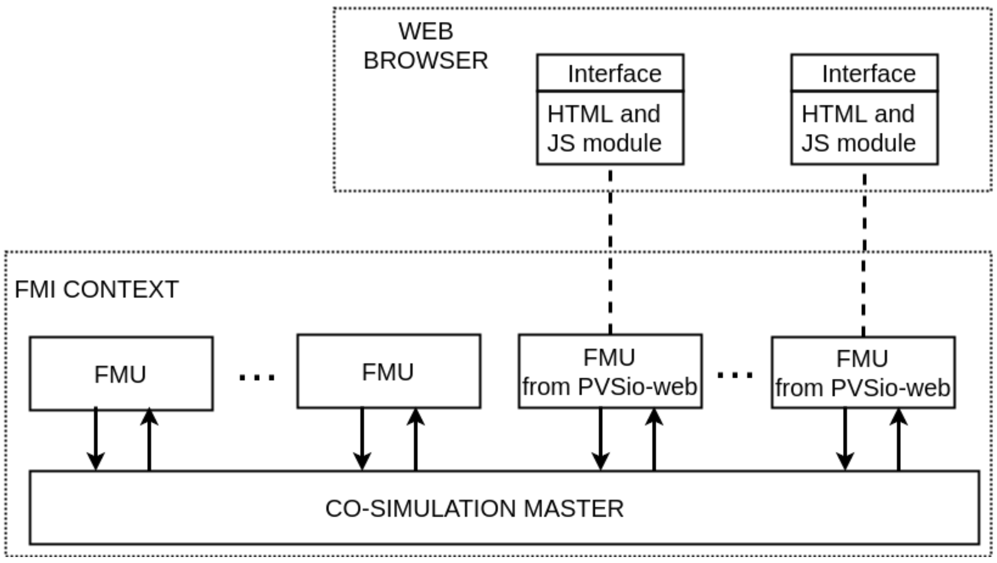


Figure 4.4: Co-simulation with FMUs of PVSio-web prototypes.

cesses: `FMI process` is for co-simulation steps, and `GUI process` is for user actions. Both processes use the PVSio environment to compute a new state of the prototype.

To enhance the flexibility of the FMUs generated by the framework, the set of input buffers of the FMU is extended with additional variables, named *external variables*. These variables store information exported by other FMUs that should be rendered on the front-end of the prototype but is not necessary for the PVS model. External variables are updated by the master algorithm with `fmi2Set()` when the standard set of FMU variables is also updated. Two examples of external variables are the x and y coordinates of the line follower robot (see Section 6.1), as they are produced by the plant FMU but the PVSio-web prototypes needs them to render the movement of the robot.

Communication between FMU and the user interface

Communication between the FMU and the user interface of the prototype is event-based. When the user performs an action on an input widget of the interface of the prototype, a JavaScript module executed on the front-end of the prototype sends a message to the FMU with information about the action that has been performed (e.g., button *up* clicked). The FMU computes the necessary state updates and sends the new state to the front-end. When the front-end receives the new state, it updates the output widgets so that the visual aspect of the front-end correctly mirrors the new state of the prototype. Besides events generated by user actions, an automatic event `refresh` is periodically sent by the front-end to the FMU, to ensure that feedback on the user interface is not stale (this may happen, e.g., when user actions are not performed for a period of time and co-simulation events change the state of the prototype during this period).

The communication pattern below is followed for handling communication between the FMU with the front-end of the prototype:

1. Upon receiving an event from the front-end, wait for permission to access the shared elements of the FMU in mutual exclusion;

2. If the event is a user action, evaluate the function associated with the user action (e.g., *click_up*). The argument of the function is the current state of the prototype. The evaluation result is the new state of the prototype after executing the action. If the event is a simple `refresh` it means that there is no user action to perform.

3. Create a message to be sent to the front-end of the prototype. The message contains the new state of the prototype (plus external variables received from other FMUs, if they are used in the co-simulation);

4. Release mutual exclusion;

5. Send the state update to the front-end.

It is important to note that user actions do not change the FMI buffers — these buffers can be updated only by `fmi2DoStep` when the master algorithm triggers the execution of a co-simulation step.

A convention is used for advancing time in the PVS specification: time is advanced by a specific function (`tick` in our model) that can be invoked only by the master algorithm. All other functions, including those representing user actions, model instantaneous changes in the system state. This is necessary to ensure the correct time synchronization between FMUs.

4.3 Verification

Formal proofs can be used to verify the critical properties of the model under analysis. In particular, the formalization of properties involves translating the formulation of the property given in natural language text into a formula that captures the intended meaning of the property. Once generic concepts defined in the property and concrete behaviors of the modeled component are linked, the property can be written into logic expressions using the PVS language. The property can then be checked by writing a PVS *theorem* that represents the property.

In particular, an invariant is a property that must hold for all states of all possible execution traces. An execution trace of the system is a sequence of states that starts with an initial state on which the state-transition function is iteratively applied to generate subsequent states. The following function `kth_step` can be used to define all sequences of states given by all possible executions:

```
kth_step(K: nat): RECURSIVE State =
  IF (K = 0) THEN init_state
  ELSE tick(kth_step(K-1))
  ENDIF
  MEASURE K
```

Function `kth_step` takes parameter `K` and recursively applies `K` steps of the state-transition function `tick`. In PVS, the termination of the recursion has to be demonstrated, and the `MEASURE` part provides such information to the type checker and prover. Since `tick` does not change the input variables, the theorem prover considers that the inputs variables can have any possible value after its invocation; therefore, the invariants are proved against all possible inputs. Co-simulation provides means to choose which property is worth proving.

Function `kth_step` is convenient when proving invariants since it allows building the proof by induction on the length of the trace. For instance, the theorem `TH1` below represents the proof of an invariant `P` while the theorem `TH2` represents the proof of the same invariant starting from a specific step `S`: The theorem prover of PVS provides support for different induction schemes, e.g., classical induction, or structural induction on graphs and paths.

```
TH1: THEOREM
  FORALL(K: nat):
      P((kth_step(K))

N: above(1)
TH2: THEOREM
  FORALL(K: above(S+N)):
  P((kth_step(K))
```

Another possible strategy for properties formalization is the following:

```
TH3: THEOREM
  FORALL(st: State):
      P((tick(st))
```

which can be used to prove a property of the state after the tick function. Using the FORALL statement implies that the property must hold for every possible combination of outputs, inputs, and parameters. Users can add constraints to the elements of the state; for example, an alternative to the previous theorem schema is the following:

```
TH3v1: THEOREM
  FORALL(st: State):
       st'parameter1 < 5 IMPLIES P((tick(st))
```

which will prove the property P only for the states with parameter1 less than 5. The same constraint is applicable to TH1 or TH2. Co-simulation provides means to assess the behavior of the whole system when the constraint is met.

Abstract model of the plant

Users can enable the verification of the whole system (the plant and the controller) by expressing the plant model as another executable PVS theory, usually a high-level representation of the plant. The high-level representation of the plant must contain enough details to keep the difference between the original and the abstract systems below an acceptance threshold of tolerance for the properties.

Co-simulation provides a means to assess this difference, as the high-level representation of the plant is an executable theory and, therefore, can replace the detailed model of the plant. The traces of the co-simulation with the original plant and the one with the high-level representation can be compared to assess the difference in the same environment. Further details will be provided in the Line Follower Robot case study.

4.4 Alternative modeling for simple CPS

PVSio-web simplifies and enhances the usage of formal methods by exploiting a graphical editor, which allows users to define models of simple reactive systems as an Emuchart diagram and then verify the models by translating the Emuchart model into a PVS theory on which the theorem prover of PVS can be applied. An example of Emuchart diagram is shown in Figure 4.5 where off and on are the operational modes of a medical device (off is the initial mode). Emuchart transitions are triples:

$$label\ [condition]\{action\}$$

where label is the name of the transition, condition is a boolean expression which enables the firing of the transition, and action is the operation which is executed.

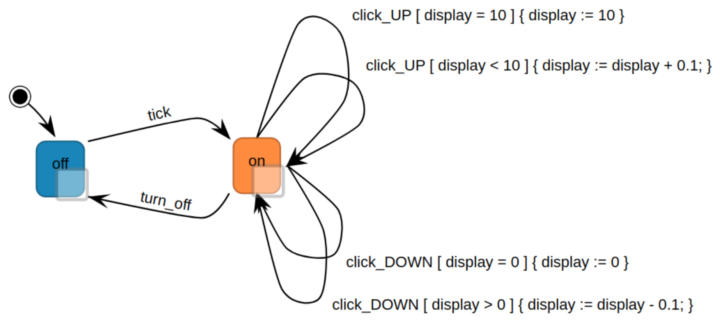

click_UP [display = 10] { display := 10 }
click_UP [display < 10] { display := display + 0.1; }
tick
off
on
turn_off
click_DOWN [display = 0] { display := 0 }
click_DOWN [display > 0] { display := display - 0.1; }

Figure 4.5: Example of Emuchart diagram.

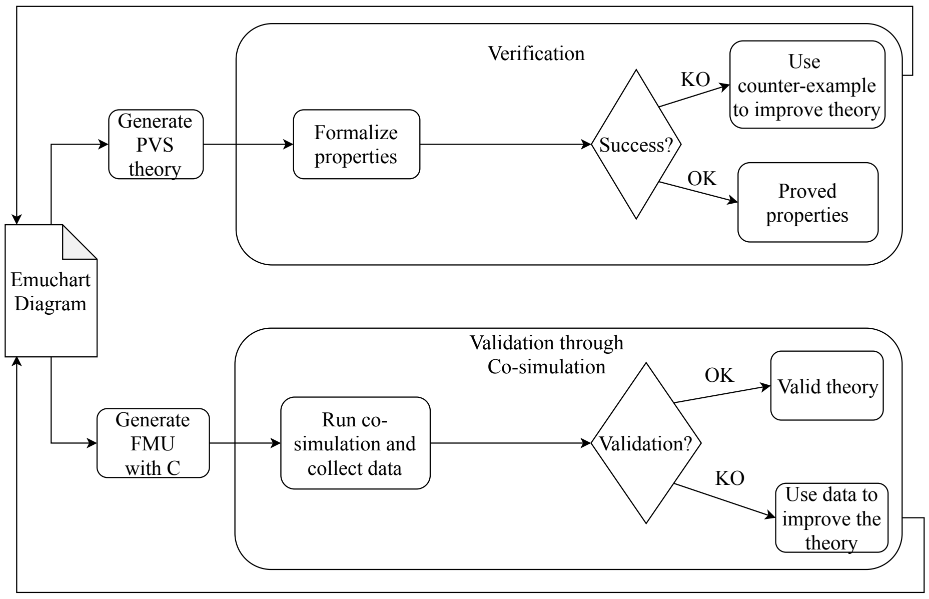


Figure 4.6: Alternative approach for simple systems.

PVSio-web also provides an automatic procedure to translate the Emuchart diagram into a MISRA C program (Mauro et al., 2017) that can be exploited to generate an FMU. The automatic MISRA C generation suggested another viable approach to integrating formal verification in co-simulation: the alternative approach is described in Figure 4.6.

The structure of the FMU generated with MISRA C is similar to the one with PVS; instead of calling the PVSio environment, the FMUs calls the function of the generated MISRA C code. The automatic generation of MISRA C code only supports simple control algorithms, because of some tools limitations, so this alternative approach is only viable for simple components. The properties verified on the PVS model derived from an Emuchart can be assumed for free in the behavior of FMU generated from the same Emuchart, assuming that the translations are well defined. An example of such properties is the deterministic behavior of the system, which is often required in many cases (e.g., critical cyber-physical systems). The type-check of the PVS prover automatically proves that when invoking a PVS function, for any possible state, (i) at least one execution path is executed (*Coverage condition*) and (ii) no more than two execution paths are executed (*Disjointness of condition*) (Owre et al., 1999). These two properties together imply the determinism of the model. As a consequence, the alternative process proves the determinism of the FMU and does not require any specific knowledge of the PVS prover.

Chapter 5

Framework for automatic integration of logic specification in FMI

The framework presented in this thesis allows developers to extend standalone PVSio-web prototypes with an FMI-2 compliant co-simulation interface. That is, given a prototype created with PVSio-web, developers can use the framework to generate an FMU that includes:

- The PVS model of the prototype specifying its behavior;
- The PVSio environment necessary for executing the PVS model;
- The XML description file used in the FMI-based co-simulation 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 (*optional*);
- A module defining the graphical front-end of the prototype (*optional*).

5.1 Implementation of the framework

The framework is implemented in JavaScript. The Handlebars[1] engine is used to generate the C source code of the FMUs. The engine supports semantic templates with *parameters* and *helper functions*. Template parameters are instantiated at run-time, using the information contained in JSON objects. Helper functions enable conditional compilation and iteration over arrays. The advantage of using semantic templates is that they are human-readable. This makes it easier for developers to check the structure of the source code by inspecting the templates — the template looks like a source code with parameters. It makes it also easier to maintain and

[1] `https://handlebarsjs.com`

```
fmi_module.create_FMU("alaris", {
  fmi: [ { name : "infusionrate",
           type : "number",
           variability: "discrete",
           scope:"local", value: "0" },
             ... ],
  init: "init_alaris",
  tick: "tick" });
```

Listing 5.1: Generation of an FMU using the APIs of the framework.

update the templates, e.g., to adapt code generation to future versions of the FMI standard or for different platforms. The same approach has been used in (Mauro et al., 2017) for generating MISRA C code for PVSio-web prototypes.

APIs provided by the framework

The APIs provided by the framework include functionalities for generating the XML description file and the C code for the FMU. The principal function is `create_FMU`. An example use of `create_FMU` is shown in Listing 5.1. The first argument (`alaris`) is a string defining the name of the FMU that will be generated. The second argument is an object defining various FMU parameters, including the list of co-simulation variables, the function in the PVS specification for initializing the prototype, and the function in the PVS specification for advancing time.

Handling of co-simulation steps

The code shown in Listing 5.2 presents the logic of the FMU module that handles the execution of a co-simulation step (`FMI process` in Figure 4.3). The important aspect to note here is that the function uses a watchdog mechanism to monitor the execution of `doStep`: if the watchdog is not reset by `doStep`, then the FMU encountered a problem (e.g., the simulation environment crashed). In this case, the FMU reports an error to the master algorithm, so that appropriate actions can be taken and the co-simulation can be continued or terminated gracefully.

```
fmi2Status fmi2DoStep(/*...function arguments omitted for brevity */) {
        doStep();
        if(watchdog == 0) { return fmi2OK; }
        return fmi2Error;
}
```

Listing 5.2: Function `fmi2DoStep`.

Relevant aspects of the implementation of function `doStep` are shown in Listing 5.3. Three main operations are performed: input variables of the FMU are read, a co-simulation step is executed, and output variables of the FMU are updated. All these operations are guarded by a lock mechanism to guarantee mutual exclusion.

```
void doStep() {
  // wait lock for mutual exclusion
  pthread_mutex_lock(&mutex);

  // read input variables
  // ... code omitted for brevity

  // execute a simulation step
  watchdog = 0;
  strcpy(previous_state, state); // save state before step
  alarm(MAX_TIME);               // set alarm of MAX_TIME seconds
  PVSioAdvanceTime();            // advance time
  if(PVSioPrint() == true){      // wait for output ready
      ualarm(0,0);               // disable alarm
      if(watchdog == 1) {        // watchdog not reset indicates error
        strcpy(state, previous_state);  // restore last valid state
        restart_PVSio();                // restore PVSio
      }
  } else { strcpy(state,temp_state); }

  // update output variables
  // ... code omitted for brevity

  // unlock of mutual exclusion
  pthread_mutex_unlock(&mutex);
}
```

Listing 5.3: Relevant fragments of function `doStep`.

Handling of user actions

A service daemon (`GUI process` in Figure 4.3) is used in the FMU to handle communication with the front-end of the prototype. The daemon invokes a specific callback handler every time an event is received on the WebSocket connection that links the FMU to the front-end of the prototype.

Listing 5.4 shows the definition of the callback handler. The behavior is as follows. Upon receiving an event from the front-end, the PVSio process is invoked to compute the new system state. If the evaluation is not completed within a given timeout (MAX_TIME), the evaluation is aborted, i.e., the state of the FMU is restored to the previous valid state, and the PVSio execution environment is restarted.

5.2 Verification of the functionalities of the framework

This Section presents the analysis of two properties of the framework:

- *Correct Execution*: FMUs generated by the framework do not block the master algorithm when user actions are received from the front-end, and a co-simulation step needs to be performed.

```
int callBack(/* ...function arguments omitted for brevity */) {
  // wait lock for mutual exclusion
  pthread_mutex_lock(&mutex);

  // check if user action has been received
  if(strcmp(in, "refresh") != 0){
    watchdog = 0;
    strcpy(previous_state, state);// save state
    alarm(MAX_TIME);              // set alarm to MAX_TIME
    PVSioEval(in);                // execute user action
    if(PVSioPrint() == true){     // wait for output ready
       ualarm(0, 0);              // disable alarm
       if(watchdog == 1){         // watchdog not reset indicates error
         strcpy(state, previous_state); // restore state
         restart_PVSio();               // restore PVSio
       }
    } else {
       strcpy(state, temp_state);
    }
  }

  // update external variables
  retrieveExternalVariables(externalvariables);
  strcpy(extendedstate, state);
  strcat(extendedstate, externalVariables);

  // unlock mutex
  pthread_mutex_unlock(&mutex);

  // send new state to the user interface
  // ...code omitted for brevity

  return 0;
}
```

Listing 5.4: Callback function used by the GUI process.

- *Correct Communication*: All state changes computed during a co-simulation step are communicated to the front-end of the prototype.

Timed Automata (Alur and Dill, 1994) are used to prove the core functionalities of the framework formally. The properties to be proved are expressed in temporal logic. A model checker tool is used to derive proofs automatically. The theory of timed automata implemented in the UPPAAL model checker (Behrmann et al., 2006) is used, and properties are expressed in timed computational temporal logic (TCTL) (Henzinger et al., 1994).

The presentation proceeds as follows: first, background information on timed automata is introduced; then, a timed automata model of the FMUs generated by the framework is presented; then, in Section 5.2, the two correctness properties (Correct Execution and Correct Communication) are verified in UPPAAL.

Background on Timed Automata

A *Timed Automaton* is a graph characterized by (i) a finite set of *locations* with one *initial* location; (ii) a finite set of *edges* connecting locations; (iii) a finite set of *actions*; (iv) a finite set of variables over the non-negative reals, called *clocks*; and (v) a finite set of predicates on clock values, called *constraints*. Each edge can be labeled with one action, a constraint, and the reset of zero or more clocks. A reset is a clock to be set to zero. The *state* of a timed automaton is given by the location and the values of the clocks at a given time.

A timed automaton models a system operating in a number of distinct *modes*. Each mode is represented by a location. Mode changes occur as a consequence of the execution of actions. While the system remains in a given location, the progress of time is reflected by the values of the clocks, whose values increase all at the same rate.

A set of timed automata can be composed to create *Networks of Timed Automata*. This is useful to build complex models using simpler models as building blocks. The synchronization between two automata is modeled by the existence in the network of two edges, one for each of the two automata, labeled with *complementary* actions, named *input* and *output* actions.

One timed automaton executing an output action synchronizes with one or more timed automata, each executing the complementary action. Two edges labeled with complementary actions can be taken only if the constraints on each of them, if any, are satisfied. Actions not participating in synchronizations, i.e., *internal* actions, are all equivalent with respect to the network behavior, and are represented by the τ action. Graphically, an input action is denoted by a question mark (*?*), an output action by an exclamation mark (*!*). The reader can refer to (Alur and Dill, 1994) for additional details on timed automata.

In the developed model, the following additional features of UPPAAL (Behrmann et al., 2006) are used: (i) integer-valued constants; (ii) bounded integer variables; (iii) *committed locations*, denoted by the letter `c`, that can be used to force no delay in a location; (iv) *urgent edges*, that use urgent channels for synchronization— when synchronization is enabled, a delay is not allowed; urgent edges cannot have clock guards, and are declared by prefixing the declaration with the keyword urgent; and (v) *location invariants*, i.e., constraints that must hold while an automaton is in a given location.

As an example, consider the timed automaton shown in Figure 5.1(b). The automaton operates on one clock `tr` and a variable `n`. When the clock is equal to the constant `deltaR`, an out edge is executed (`tr <= deltaR` is the state invariant). If variable `n` equals to `0`, the edge that is executed sends a `refresh` message and resets the clock. Otherwise (`n > 0`) the edge that is executed only resets the clock. Figure 5.1(c) shows an example of synchronous action (`refresh`) between `GUIrefresh` and Buffer. The action increments variable n if the value of n is not already the maximum value.

Timed automata model of FMUs generated by the framework

In this Section, a network of timed automata is developed for representing relevant structural elements of the FMUs described in Section 5.1. The developed model is shown in Figure 5.1. It includes eight timed automata. The behavior of the front-end of the prototype (hereafter, *GUI*) is modelled by User, GUIrefresh, Buffer and WebsocketClient. The behavior of the back-end of the prototype (hereafter, *FMU*) is modeled by WebsocketServer, Mutex, FMU and Master.

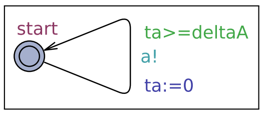
The set of actions is:

$$\mathcal{A} = \{\texttt{a}, \texttt{refresh}, \texttt{doAction}, \texttt{actionDone}, \texttt{lockRequest}, \texttt{lockGranted}, \texttt{lockRelease}, \texttt{fmi2DoStep}, \texttt{endStep}\}$$

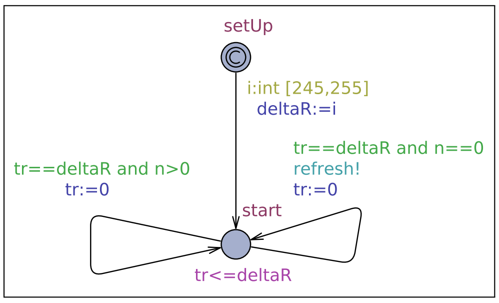
A global variable n is defined in the model, which counts the number of messages buffered in the communication link between GUI and FMU. This variable is initialized in the Buffer automaton. The following constants and parameters are defined in the model:

- deltaA, the minimum interval between two user actions
- deltaC, the maximum interval spent in the critical section
- deltaM, the minimum interval between two co-simulation steps (fmiDoStep)
- deltaR, the maximum interval of time between two updates of the GUI

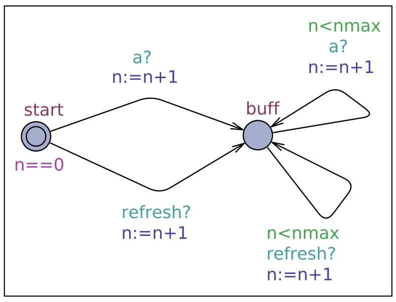
The following clocks are specified in the model: $Clk = \{\texttt{ta}, \texttt{tc}, \texttt{tm}, \texttt{tr}\}$. A global variable c is shared between WebsocketServer and FMU. The variable is initially set to 0. When WebsocketServer exits the critical section, action lockRelease in WebsocketServer assigns 1 to c. When FMU exits the critical section, action lockRelease in FMU assigns 2 to c.



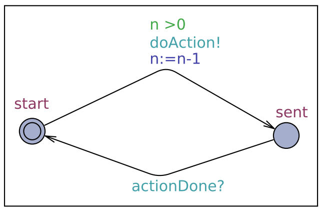
(a) User.



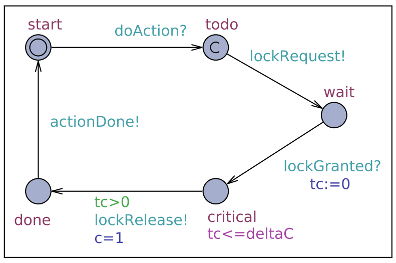
(b) GUIrefresh.



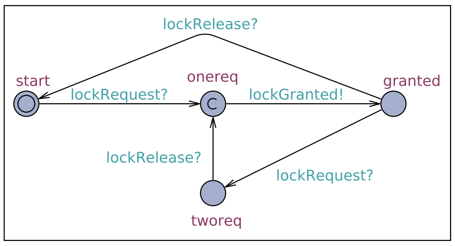
(c) Buffer.



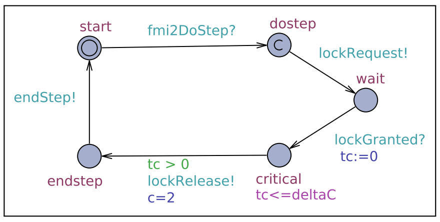
(d) WebsocketClient.



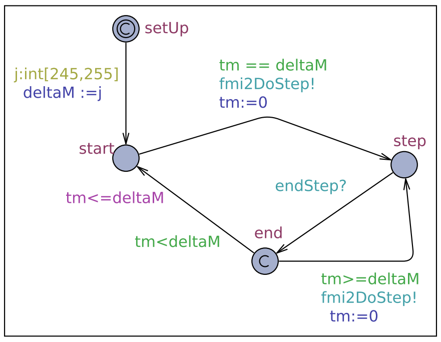
(e) WebsocketServer.



(f) Mutex.



(g) FMU.



(h) Master.

Figure 5.1: Timed automata model of an FMU.

The roles played by the automata are as follows:

- `User` models actions performed by the user on the GUI (`a!` represents a generic user action on input widgets).

- `GUIrefresh` models the automatic refresh actions performed by the GUI every `deltaR` time unit. The initial state of this automaton is a `committed` state. Initially, a random value in the interval $[245, 255]$ms is assigned to deltaR. This value represents a refresh rate of 4 Hertz, which is a reasonable value for a standard user interface, and a clock variability of +5/-5 milliseconds, which takes into account possible delays/imperfections in the clock.

- `Buffer` models the buffering of actions at the GUI. The content of messages is abstracted in the model — the automaton only counts the number of buffered actions. Upon receiving `a?` or `refresh?`, the global variable `n` is incremented. The initial state satisfies the invariant `n == 0`.

- `WebsocketClient` handles communication from GUI to FMU. If the buffer contains messages (i.e., `n > 0`), `doAction!` is executed and `n` is decremented. Communication is synchronous. From state `sent`, the automaton moves to `start` upon executing the synchronisation action for WebSocket communications (`actionDone?`). The model assumes that messages in the buffer are processed immediately. Both `doAction!` and `actionDone?` are urgent actions, i.e., when synchronization is enabled, the actions are executed immediately.

- `WebsocketServer` handles communication from FMU to GUI. Upon receiving an action (`doAction?`), a critical section is entered, PVSio is invoked for evaluation of the action, and the result of the evaluation is sent back to the GUI (`actionDone!`). A lock is requested to access the critical section (`lockRequest!`). After lock is granted (`lockGranted?`), the automaton exits the critical section in less than `deltaC` time units (invariant `tc <= deltaC` on state `critical`) by executing `lockRelease!`. When `WebsocketServer` exits the critical section, variable c is assigned 1.

- `Mutex` models the mutual exclusion mechanism for accessing the critical section. Lock requests (`lockRequest!`) received when the critical section is busy (state `tworeq`) are granted when the critical section is released.

- `FMU` models the execution of a co-simulation step. Upon receiving action `fmiDoStep?`, input variables are read, PVSio is invoked for evaluation of the step, and the result of the evaluation is used to update the output variables. These operations are carried out in the critical section, and are protected by a lock request (`lockRequest!` and `lockGranted?`). The lock on the critical section is released in *deltaC* units (`lockRelease!`). Upon exiting the critical section, the variable *c* is assigned 2, and the successful execution of the step is communicated with `endStep!`.

- `Master` models the master algorithm of the co-simulation. The UPPAAL model is constructed based on the specification of the FMI protocol. The co-simulation engine of INTO-CPS follows the standard. The developed model includes only the details necessary to verify the considered properties. The execution of a co-simulation step is requested every *deltaM* time units with action `fmiDoStep!`. *deltaM* is a random value in the interval $[245, 255]$, so that some properties can be proved related to the condition $deltaR < deltaM$. The initial state is a `committed` state. When the execution of the co-simulation step completes, action `endStep?` is triggered. Two cases are considered. If the execution of the co-simulation step took less than *deltaM* (i.e., `tm < deltaM`), the automaton moves to state `start`, that will be exited at `deltaM`, for a new co-simulation step. Otherwise, a new co-simulation step is executed and the automaton moves to state `step`. Clock `tm` is reset at the beginning of each step. Action `endStep?` is an urgent action.

Analysis of correctness properties of the framework

Simulation and formal verification were used to perform the analysis of correctness properties of the framework. The analysis is performed under the following timing assignment: `deltaA = 10ms`, `deltaR` and `deltaM` in the interval `[245, 255]ms`, `deltaC = 1s`. These are possible values for the co-simulation of systems with user interfaces.

Analysis of Correct Execution. Analysis of this property is carried out using simulation. The sequence diagram in Figure 5.3 illustrates part of a simulation run in UPPAAL. The considered case is when `WebsocketServer` and `FMU` want to access shared resources. The following steps are performed:

1. Assignment of a random value to `deltaR` (transition in `GUIrefresh` from `setUp` to `start`) and assignment of a random value to `deltaM` (transition in `Master` from `setUp` to `start`);

2. Execution of a `refresh` action, represented by a sequence of synchronizations from `refresh` to `actionDone`;

3. Execution of a user action `a`;

4. Execution of a co-simulation step (synchronization `Master` - `FMU`);

5. Lock of the critical section acquired by `FMU` process;

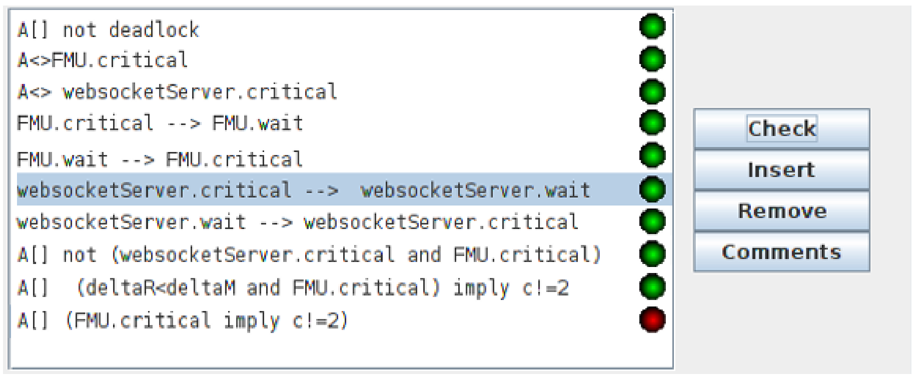

```
A[] not deadlock
A<>FMU.critical
A<> websocketServer.critical
FMU.critical --> FMU.wait
FMU.wait --> FMU.critical
websocketServer.critical --> websocketServer.wait
websocketServer.wait --> websocketServer.critical
A[] not (websocketServer.critical and FMU.critical)
A[] (deltaR<deltaM and FMU.critical) imply c!=2
A[] (FMU.critical imply c!=2)
```

Figure 5.2: Verification of *Correct Communication* in UPPAAL.

6. User action sent from `WebsocketClient` to `WebsocketServer`;
7. Lock request sent by `WebsocketServer`;
8. `WebsocketServer` waits lock release.

Analysis of Correct Communication. An analysis of this property is carried out using verification. Figure 5.2 shows the formulae used to check the core properties of the FMUs. Properties are proved under the following constraints:

- Co-simulation steps are executed every `deltaM` time units, or as soon as the previous co-simulation step terminates;
- Communication between GUI and FMU is synchronous, i.e., GUI waits for a response from FMU after sending an evaluation request;
- A new message is stored in the WebSocket buffer at most every `deltaR` time unit;
- The WebSocket buffer has a finite length.

At the implementation level:

- The assumption of bounded time spent in the critical section is guaranteed by the watchdog mechanism (maximum duration of the critical section equals to the watchdog timer `tc` $\leq$ `deltaC`, where `deltaC` = MAX_TIME)
- All PVS functions are guaranteed to terminate: this can be checked by inspecting the *termination conditions* generated by the PVS type checker.

The first formula in Figure 5.2 (`A[] not deadlock`) is a particular case in UPPAAL for proving that for each reachable state, there exists at least one outgoing edge. The formula is satisfied for all states; therefore, the formula above guarantees the absence of deadlock in the system.

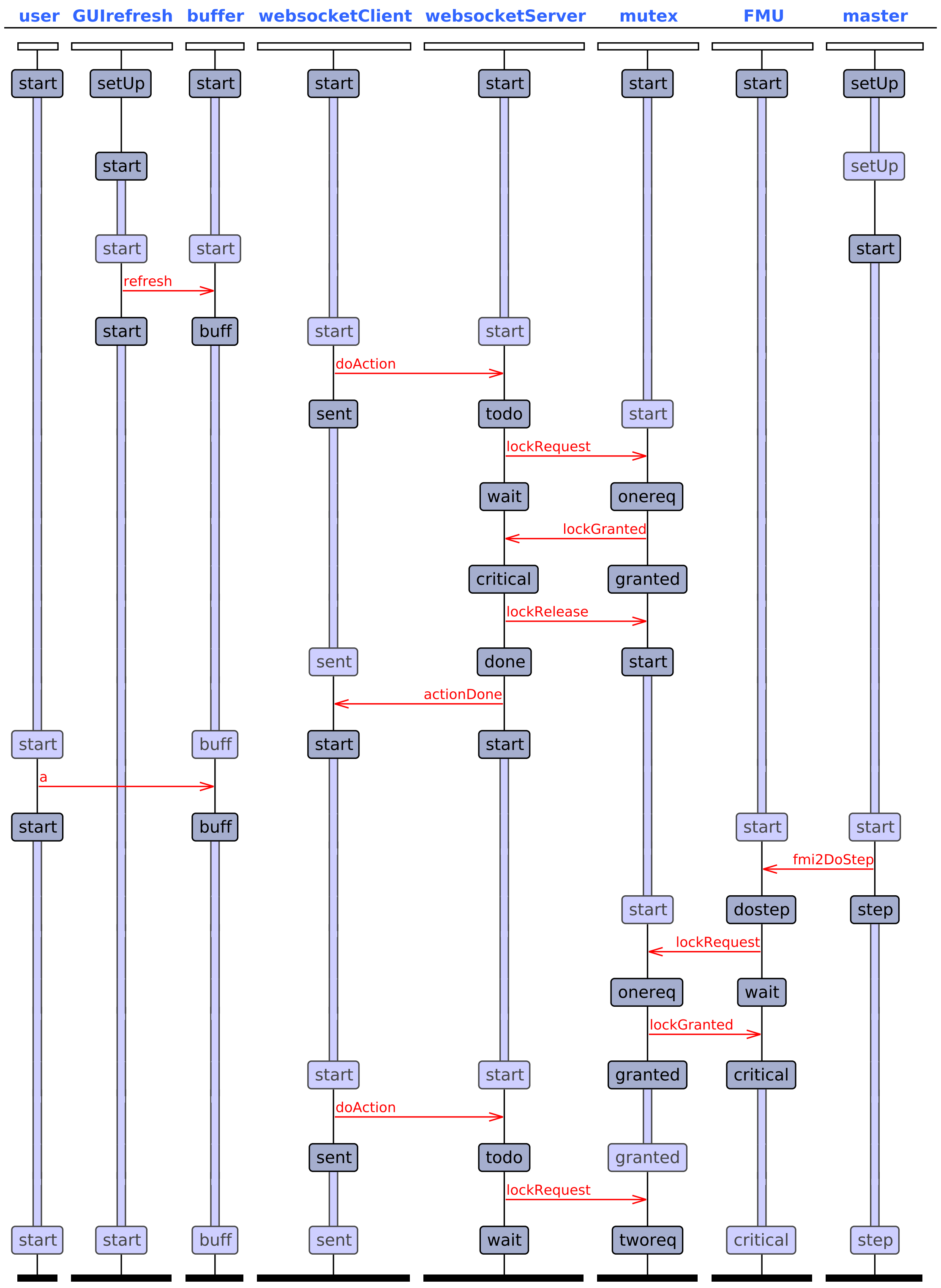


Figure 5.3: Analysis of *Correct Execution* in UPPAAL.

The following two formulae prove that for all execution paths, `WebsocketServer` and `FMU` eventually gain access to the critical section. This is an important functional aspect of the framework that relates to fairness of the execution of the two processes, `FMU`, and `WebSocketServer`. This is enforced by the following fairness policy: if both `FMU` and `WebSocketServer` requests access to the critical section and `FMU` was the last process that was granted access to the section, then `WebSocketServer` will be granted access to the critical section. The same also applies the other way around.

The following four formulae are liveness properties. They prove that: when the `FMU` is in the critical section, it will eventually exit; when the `FMU` waits for the critical section, it will eventually enter. The same properties are also proved for `WebsocketServer`.

Formula `A[] not (websocketServer.critical and FMU.critical)` is a safety property. It proves that, in all states of the model, it is never the case that `WebsocketServer` and `FMU` are both in the critical section.

Formula `A[] (deltaR < deltaM and FMU.critical) imply c!=2` is another safety property. It proves that, under specific timing constraints, all FMU states are correctly reported to the user interface. In particular, it is shown that, if $deltaR < deltaM$, any two executions of `FMU` in the critical section are always interleaved by at least one execution of `WebSocketServer` in the critical section. This guarantees that any state change of the FMU can be displayed on the GUI. The random value assigned to *deltaR* and *deltaM* forces the model checker to explore all the possible combinations of values.

The last formula (`A[] (FMU.critical imply c != 2)`) shows that without the condition $deltaR < deltaM$, the previous formula is falsified. In this case, more than one co-simulation steps can be executed between two updates of the GUI.

Discussion

In the FMUs, time is not advanced when a user action is performed. Rather, time is advanced by the master algorithm, when `doStep` is executed. This is necessary to comply with the FMI co-simulation protocol, which requires time to be advanced only by `doStep`.

User actions are asynchronous with respect to the co-simulation step. This is necessary to produce prototypes that provide a consistent response time to user actions during co-simulation runs. This approach does not affect the correctness of the FMU with respect to the FMI protocol, because output buffers of the FMU are only updated when `doStep` is executed.

When decomposing an FMU into smaller FMUs, co-simulation parameters need to be chosen carefully. Different decompositions may impose different constraints on the exchange of co-simulation events. For example, if it is essential to communicate all user actions received by the FMU of the PVSio-Web prototype to the other

FMUs, the co-simulation step needs to be shorter than the minimum time between two user actions. On the other hand, if the FMU of the PVSio-Web prototype represents a standalone system and user actions are communicated to other simulated components only at given instants of time, this constraint between the interval of co-simulation step and minimum time between user actions does not apply.

Finally, it is worth noting that the presented formalization of the framework focuses on functional correctness. Performance aspects of the framework, such as responsiveness of the co-simulation in the case of a rapid sequence of user inputs or rapid co-simulation steps, are not considered here and need further investigation.

Chapter 6

Case studies

This chapter shows the application of the methodology and the framework to case studies from different domains:

- a simple semi-autonomous vehicle, the Line Follower Robot (LFR), with manual/automatic control mode. The controller is modeled in PVS, while the sensors and the robot are modeled with other tools. The experiments are run using the INTO-CPS application and the analysis in case of faults to the sensors or to the robot is reported. Finally, a property of the complete system is shown by defining a PVS theory of the robot.
- an Unmanned Aerial Vehicle (UAV) coordination protocol. A high level distributed coordination algorithm commands a lower level flight controller to achieve a desired configuration of a drone swarm. Co-simulation is used combining different simulators for the drone dynamics (OpenModelica (Fritzson et al., 2005)), the flight control (C program), and the coordination algorithm (PVS).
- an Integrated Clinical Environment (ICE) composed of an infusion pump, a patient monitor and a remote supervisor device. All the medical devices are modeled with PVSio-web and they are co-simulated together with a pharmacokinetics model of the patient modeled with OpenModelica. The introduction of the patient model and the human-in-the-loop co-simulation made it possible to gain useful insights on the PVSio-web models. Finally the theorem prover is used to prove safety properties of the pump component.

6.1 Line Follower Robot

The Line Follower Robot case study, provided by the INTO-CPS (Larsen et al., 2016), consists of an autonomous robot with the task of following a line painted

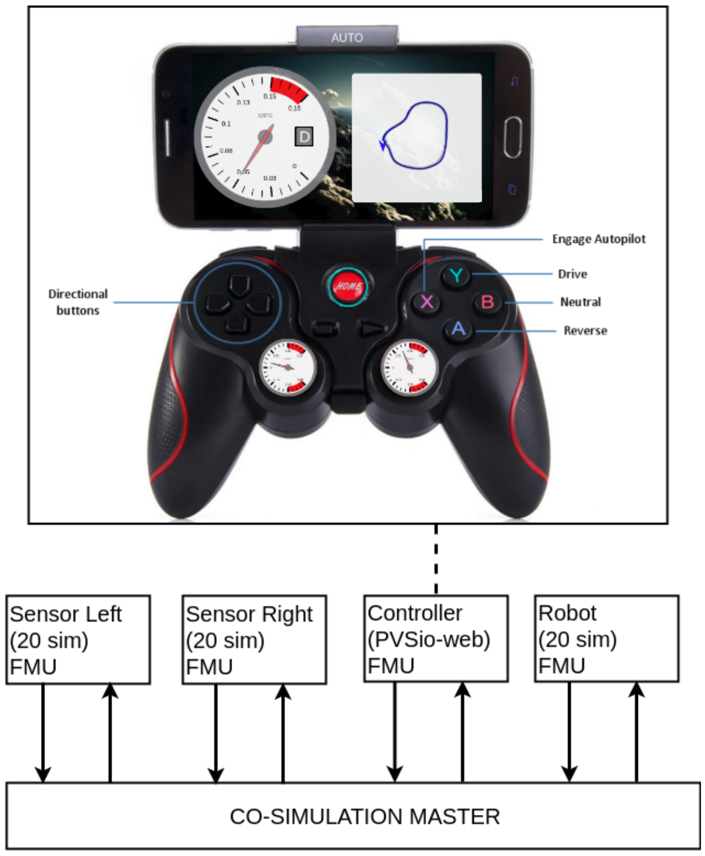


Figure 6.1: Co-simulation of line follower robot case study.

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 20-sim (Broenink, 1999)), the FMU of the sensors (created with 20-sim or OpenModelica), and the FMU of the controller (created with Overture (Larsen et al., 2010)).

In this thesis, the controller has been modeled as a PVS theory and a new advanced controller that allows manual override of the automatic line following behavior has been developed. The robot can be controlled using a joypad (see Figure 6.1) that 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 (*up arrow*) or brake (*down arrow*), change direction of the robot (*left and right arrows*), and change gear (buttons *A, Y and B*). Pushing any button on the joypad changes the control mode from automatic to manual. There is also a control (button *X*) to switch control mode from manual back to automatic.

```
LFR_controller_th: THEORY BEGIN IMPORTING types_and_constants
    %-- state attributes
    State: TYPE = [#
      lightSensors: [# left: real, right: real #]
      motorSpeed  : [# left: real, right: real #],
      gear: Gear,
      cm  : ControlMode
    #]
    init_state: State = ... %-- omitted for brevity

    %-- transition functions
    accelerate(st: State): State = st WITH [
      cm := MANUAL,
      motorSpeed := (#
        left := COND
               gear(st) = DRIVE ->
                 inc_CW_speed(motorSpeed(st)`left, ACC_STEP),
               gear(st) = REVERSE ->
                 inc_CCW_speed(motorSpeed(st)`left, ACC_STEP),
               gear(st) = NEUTRAL -> motorSpeed(st)`left ENDCOND,
        right := COND
               gear(st) = DRIVE ->
                 inc_CCW_speed(motorSpeed(st)`right, ACC_STEP),
               gear(st) = REVERSE ->
                 inc_CW_speed(motorSpeed(st)`right, ACC_STEP),
               gear(st) = NEUTRAL -> motorSpeed(st)`right ENDCOND #)
    ]
    tick(st: State): State = .... %-- omitted for brevity
    %... more definitions omitted
END LFR_controller_th
```

Listing 6.1: Snippet of the PVS theory of the new controller.

The state of the model is reported in Listing 6.1. It contains the following attributes:

- `lightSensors` stores the light sensor values;
- `motorSpeed` stores the control values sent to the left and right wheels motors;
- gear is the current gear (DRIVE, REVERSE or NEUTRAL) of the robot;
- `cm` is the current control mode (AUTO or MANUAL).

An example transition function, `accelerate`, is shown in Listing 6.1. The function is invoked by pressing the *up arrow* and it sets the control to `MANUAL` and increases the speed of the robot of `ACC_STEP` based on the current gear.

The invocation of the API of our framework for creating an FMU for this prototype is shown in Listing 6.2. Argument `fmi` includes:

- six fields necessary to store information about the attributes of the PVS state (`lightSensors` and `motorSpeed` store two values each);

- four fields for the external variables provided by the other FMUs: current position of the robot, linear, and angular speed.

```
fmi_module.create_FMU("line_following_robot",{
   fmi: [
        {name: "left", parent:"lightSensors", type:"real",
          variability: "continuous", scope: "input", value: "0"},
        {name: "right",parent:"lightSensors", type:"real",
          variability: "continuous", scope: "input", value: "0"},
        {name: "left", parent:"motorSpeed", type:"real",
          variability: "discrete", scope: "output", value: "0"},
        {name: "right",parent:"motorSpeed", type:"real",
          variability: "discrete", scope: "output", value: "0"}
        {name : "gear", type:"string",
          variability: "discrete", scope:"local", value:"0"},
        {name : "cm", type:"string",
          variability: "discrete", scope:"local", value:"0"},
        {name : "linear_speed", type: "real",
          variability: "discrete", scope:"input", value:"0"},
        {name : "angular_speed", type: "real",
          variability: "discrete", scope:"input", value:"0"},
        {name : "position_x", type: "real",
          variability: "discrete", scope:"input", value:"0"},
        {name : "position_y", type: "real",
          variability: "discrete", scope:"input", value:"0"},
   ],
   init:"init_state",
   tick:"tick"
});
```

Listing 6.2: Invocation of the API of our framework.

A snippet of the XML file generated by the invocation of the create_FMU is shown in Listing 6.3. The first part of the file is a descriptor of the FMU (model name, identifier, step size, etc). Field `ModelVariable` enumerates the variables of the FMU (see Listing 6.2). Each variable is associated with a `valueReference` necessary to find the value of the variable within the buffers of the FMU.

A snippet of the `instantiate` function generated by the invocation of the create_-FMU is shown in Listing 6.4. Lines 7-14 show the actual code generated by the Handlebar engine with the `fmi` object shown in Listing 6.2.

The `create_FMU` API also generates a custom makefile that can be used to build the FMU. The new FMU was successfully used, together with the FMUS provided by INTO-CPS, in many co-simulation scenarios orchestrated by the INTO-CPS Co-simulation Orchestration Engine. The FMU connected with the PVSio-web navigation display has been used to analyze the robot behavior when switching control mode from manual to automatic and to expose possible faults of the robot (see Figure 6.2).

Co-simulation run

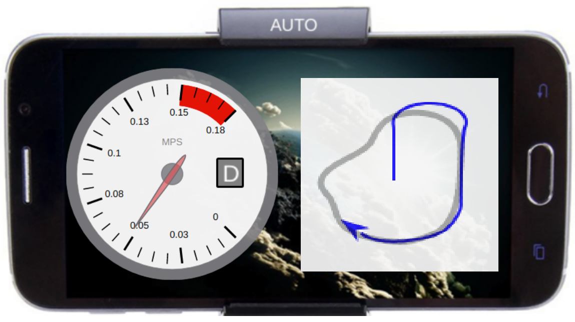
Co-simulation runs can be used to validate the advanced controller. For example, some experiments pointed out the need to perform a U-turn to get back on track when switching from manual to automatic control and the robot was moving at

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription fmiVersion="2.0" modelName="line_following_robot"
         ... >
    <CoSimulation modelIdentifier="line_following_robot"
          ... >
  </CoSimulation>
  <LogCategories> ... </LogCategories>
  <ModelVariables>
     <ScalarVariable name="gear" valueReference="1"
                     causality="local" variability="discrete">
                     <String  /></ScalarVariable>
     <ScalarVariable name="cm" valueReference="2"
                     causality="local" variability="discrete" >
                     <String  /></ScalarVariable>
     <ScalarVariable name="linear_speed" valueReference="3"
                     causality="input" variability="discrete" >
                     <Real start="0" /></ScalarVariable>
     ...
     <ScalarVariable name="motorSpeed_right" valueReference="10"
                     causality="output" variability="discrete" >
                     <Real  /></ScalarVariable>
    </ModelVariables>
  <ModelStructure> ... </ModelStructure>
</fmiModelDescription>
```

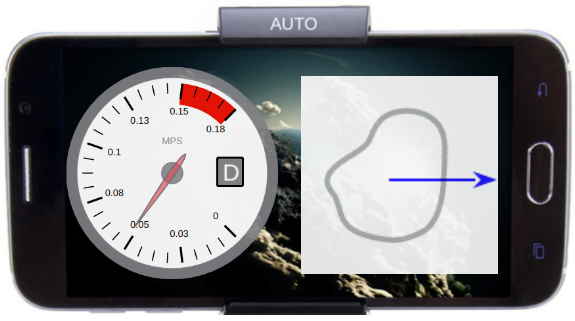
Listing 6.3: Snippet of generated XML.

```c
void instantiate(const char* location) {
    //... code for accessing the PVSio environment omitted for brevity
  start_PVSio_process(pid_PVSio_process);
  PVSioPrint();

    // update output variables
    index_state = findVariable("left", state);
    if (index_state != -1) {
        writeOutputVariableDouble(index_state, 9);
    }
    index_state = findVariable("right", state);
    if ( index_state != -1){
        writeOutputVariableDouble(index_state, 10);
    }

    open_websocket();
    pthread_create(&child1, NULL, &GUI_process, NULL);
}
```

Listing 6.4: Snippet of the generated `instantiate` function.



(a) Initial U-turn due to high speed.



(b) Missed turn.

Figure 6.2: Unexpected paths followed by the line follower robot.

high speed (see Figure 6.2a), 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 6.2b).

Simulating faults

This section shows an example of fault analysis extension introduced in Section 4.1. The extension is applied to the autonomous controller. A fault is injected into the system by executing the controller together with the functions modeling the failing behavior of the component. In order to model faults, the robot state is extended with fields characterizing the different types of faults.

In the present example, three faults are considered: a fault to sensors that occurs once and acts indefinitely; another fault to sensors that repeated N number of steps every M steps (N < M); and fault to actuators that occurs sporadically with a duration of one co-simulation step.

The following function implements the failure mode for a fault that indefinitely forces to *black* the value read by the left sensor, starting from a randomly chosen co-simulation step. Function NRANDOM in the initial state is invoked with an upper bound of 500. Variable `lightSensors` is modified (`140` is the constant for black color); clock *start_step* specifies the co-simulation step at which the fault is activated.

The failure mode is defined as:

```
fm_fault1(st: ext_State): ext_State = % sensor fault
    IF stepCounter(st) > start_step(st)
    THEN st WITH [
        lightSensors :=
            (# left := 140,
               right := st`lightSensors`right #)
        ]
    ELSE st
    ENDIF
```

The following function implements the failure mode for a fault that forces to *white* the value read by the left sensor for `L` steps every `M` timesteps. This is repeated indefinitely, starting from the first co-simulation step. Variable `lightSensors` is modified (`160` is the constant for white color); clock *elapsed_steps* specifies the number of steps since the last fault has been activated. The failure mode is defined as:

```
fm_fault2(st: ext_State): ext_State = % sensor fault
    IF elapsed_steps(st) <= L
    THEN st WITH [
        lightSensors :=
            (# left := 160,
               right := st`lightSensors`right #),
        elapsed_steps :=  elapsed_steps + 1
        ]
    ELSE IF elapsed_steps(st) < M
          THEN st WITH [
             elapsed_steps :=  elapsed_steps + 1
            ]
            ELSE IF elapsed_steps(st) = M
                 THEN st WITH [
                      elapsed_steps := 0 ]
    ENDIF
```

The following function implements the failure mode for a fault that sporadically switches off the power of each motor for one co-simulation step. The co-simulation step, at which the power of each motor is switched off, is chosen randomly. Function NRANDOM is invoked in the initial state and in the function with an upper bound of 20. Clock *occurrence_step* specifies the co-simulation step at which the next occurrence of the fault starts. The failure mode is defined as:

```
fm_fault3(st: ext_State): ext_State =  % actuator fault
   IF stepCounter(st) = occurrence_step(st)
   THEN st WITH [
        motorSpeed :=
            (# left := 0,
               right := 0 #),
               occurrence_step := NRANDOM(20) + 1
                ]
   ELSE st
   ENDIF
```

The full definition of *ext_State*, including information about the faults above, is the following:

```
  ext_State: TYPE =
       [# original_state : State
          % global clock
          stepCounter:int,
          % local clocks
          start_step: int, % fault1
          elapsed_steps: int,  % fault2
          occurrence_step: int  % fault3
       #]
```

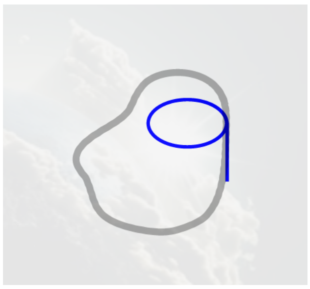
In the initial state, the step at which `fault1` starts and the the step at which the first occurrence of `fault3` starts are initialized with a random value; the elapsed steps since the last occurrence of the `fault2` is initialized to 0:

```
  init_ext_state: ext_State =
      (# original_state := init_state
        % global clock
        stepCounter := 0;
        % fault1
        start_step = NRANDOM(500),
        % fault2
        elapsed_steps = 0,
        % fault3
        occurrence_step = NRANDOM(20) + 1
       #)
```

The following example shows the `ext_tick` function for the analysis of the actuators fault previously described:

```
ext_tick(st: ext_State): ext_State =
         LET ext_state1 = st WITH (
         original_state := tick(st‘original_state),
         stepCounter := stepCounter + 1 )
         IN fm_fault3(ext_state1)
```

Execution Traces


(a) Fault1.

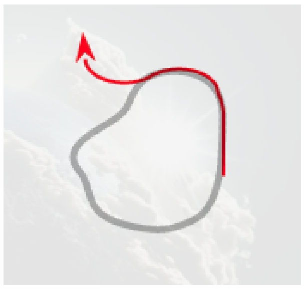

(b) Fault2.

Figure 6.3: Faults to sensors.

Figure 6.3a shows the sample trajectory for the fault1 to sensors. The left sensor is stuck at a fixed value, so that the robot starts turning at the onset of the fault, ending up in a closed trajectory.

Figure 6.3b shows the effect of the fault2 to sensors, assuming `L = 40` and `M = 2*L`. At a given point, the robot looses the line. The same fault under different values of `L` has no effect. In case of `L = 20`, the trajectory of the robot is equivalent to the one shown in Figure 6.4a. This example is better investigated in the next Section.

Figure 6.4b shows the sample trajectory when the fault3 to actuators occurs. The robot follows the nominal path, but the execution traces, reporting the simulated time at each simulation step, show that the robot is delayed with respect to the resulting trace shown in Figure 6.4a. This is expected, since the fault consists in stopping both motors for a short time. Since the motors stop at the same time, the robot heading at each instant is unchanged.

Figure 6.5 shows the two outputs of the control FMU during a co-simulation run: according to the actuator fault function the values on the graph sporadically go to zero.

Formal verification

In this section some invariants of the autonomous controller are proved.

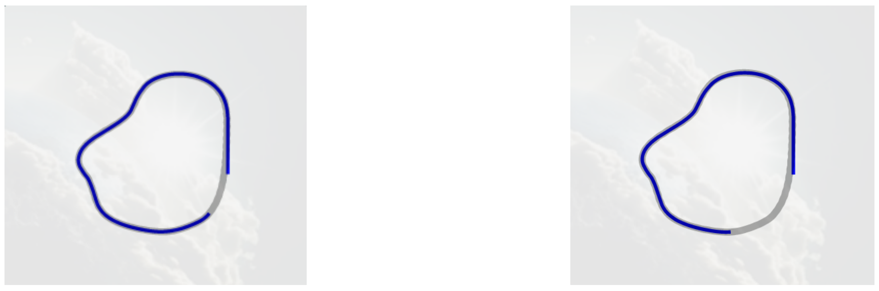

(a) No fault. (b) Fault3.

Figure 6.4: Fault to actuators.

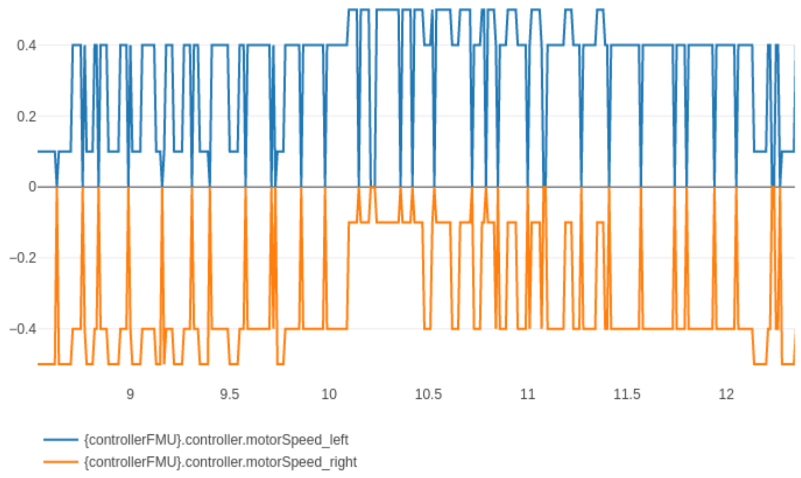


Figure 6.5: INTO-CPS graph of left motor (blue line) and right motor (red line) speed under fault3.

A property of the Controller

As an example of a possible verification on the controller's behavior, this section shows results concerning faults causing a sensor to be stuck at a given value.

In particular, the following theorem shows that, under a fault forcing the left sensor to read a black value, the robot can never turn right (i.e. motorSpeed of the right wheel will always be greater or equal than motorSpeed of the left wheel).

```
N: above(1)
never_right_random: THEOREM
    FORALL(K: above(NRANDOM(500) + N)):
        motorSpeed(kth_step(K))‘left
        <= -motorSpeed(kth_step(K))‘right
```

```

Rule? (induct K)
Inducting on K on formula 1,
this yields  2 subgoals:
never_right_random.1 :

  |-------
{1} motorSpeed(
      kth_step(NRANDOM(500) + N + 1))`left <=
     -motorSpeed(
      kth_step(NRANDOM(500) + N + 1))`right
  .
  .
  .
never_right_random.2 :

  |-------
{1} FORALL (ja: above(NRANDOM(500) + N)):
        motorSpeed(kth_step(ja))`left <=
        -motorSpeed(kth_step(ja))`right
        IMPLIES
        motorSpeed(kth_step(ja + 1))`left <=
        -motorSpeed(kth_step(ja + 1))`right
```

Listing 6.5: Application of the induction strategy.

The proof is by induction on the number of steps *K*. The prover's *induct* rule generates the induction base and the inductive step shown in Listing 6.5

Both subgoals are proved with a lengthy but obvious sequence of function expansions, introduction of a small number of intermediate lemmas (not shown), and automatic simplifications with the *simplify* rule, closed by invocations of the *assert* and *grind* rules that conclude the subproofs.

A theory for the robot kinematics

In the preceding sections, the robot kinematics have been simulated by an FMU encapsulating a 20-sim model, and formal verification has addressed only the controller model. In this section, a PVS model of the robot kinematics is introduced. This model extends the original definition of *State* by introducing new fields for co-simulation: step size (*stepsize*), linear and angular speed (*linspeed* and *angspeed)*, position (coordinates *xx* and *yy*), and direction (angle *theta*).

```
extended_robot: THEORY
 BEGIN
 IMPORTING LFR_controller
  ext_State: TYPE =
        [#  state: State,
            stepsize: real,
            linspeed: real,
            angspeed: real,
            xx: real,
            yy: real,
            theta: real
         #]
         ...
END extended_robot
```

The model also extends the previous definition of *tick* introducing two new functions: *update_position* and *update_speed*.

Function *update_speed* invokes the *tick* function and then computes the linear and angular speeds. Function *update_position* updates the position and the direction of the robot based on the linear and angular speeds, see Listing 6.6.

The formulas used in these functions are a simplified version of the formulas used in the 20-sim models. The simplification consists of introducing instant changes of both the linear speed of the robot and the angular one. The constant values in function *update_speed* are taken from a co-simulation log and they represent: the maximum linear speed when the robot is moving forward (0.062 m/s), the maximum linear speed when the robot is turning (0.057 m/s) and the angular speed when the robot is turning (0.47 rad/s).

Using the maximum speed, we obtain an over-approximation of the distance covered by the robot. This abstract model can be used to prove properties such as the one described later in this section, where the maximum distance covered in a co-simulation step is used in the proof; if this property is satisfied using the abstract model, the property holds also on the real system.

The new theory has been embedded into an FMU and co-simulated for validation: the result is shown in Figure 6.6 where the temporal evolution of the y and x coordinates generated with the 20-sim model (blue lines) are compared with the ones of the PVS model (red lines); the behavior is the same within a tolerance of approximately 5 centimeters.

PVS allows us to describe a system at different levels of abstractions. By adding more details to the PVS kinematics model, finer properties could be proved and better tolerance could be achieved.

```
ext_tick(st: ext_State): ext_State =
  update_position(update_speed(st))

update_position(st: ext_State): ext_State =
  st WITH
[
    xx:= st‘xx-st‘linspeed*st‘stepsize*SIN(st‘theta),
    yy:= st‘yy+st‘linspeed*st‘stepsize*COS(st‘theta),
    theta := st‘theta+st‘angspeed*st‘stepsize
]
update_speed(st: ext_State): ext_State =
  LET st1: ext_State =
    st WITH [state := tick(st‘state)] IN
     st1 WITH [
       linspeed := COND
    st1‘state‘motorSpeed‘left = 0 -> 0,
    st1‘state‘motorSpeed‘left = med -> 0.062,
    st1‘state‘motorSpeed‘left = hi -> 0.057,
    st1‘state‘motorSpeed‘left = low -> 0.057,
    else -> st1‘linspeed
       ENDCOND,
       angspeed := COND
    st1‘state‘motorSpeed‘left = 0 -> 0,
    st1‘state‘motorSpeed‘left = med -> 0,
    st1‘state‘motorSpeed‘left = hi -> -0.47,
    st1‘state‘motorSpeed‘left = low -> 0.47,
    else -> st1‘angspeed
       ENDCOND
       ]
```

Listing 6.6: Extended tick for kinematics theory.

A property of the complete CPS

The usage of the kinematic theory enables the proof of properties related to the physical process. Considering the following assumptions:

- `L` a value that represents a generic number of co-simulation steps
- `S` a value for the step size (*stepsize*)
- 1.5 centimeters (0.015 meters) the width of the black line painted on the floor of the robotic system.

the PVS theorem prover can be exploited to prove the following property:

> **P1**: *under the hypothesis that* `L * S <= 0.24`*, within* `L` *steps the y coordinate of the robot change at most of 1.5 centimeters (0.015 meters)*

The property can be expressed in PVS as follows:

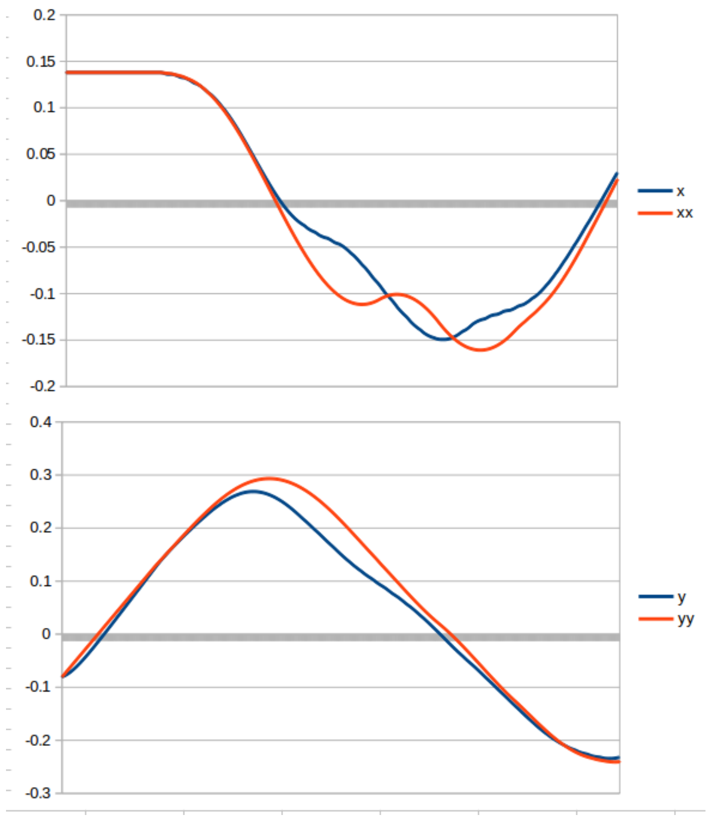


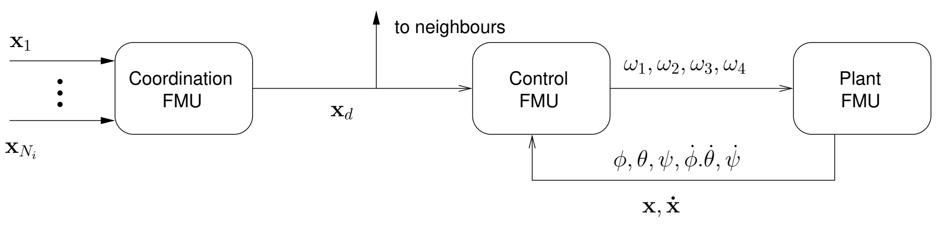
Figure 6.6: Comparison between 20-sim model and PVS model.

```
P1: THEOREM
 L*S <= 0.24 IMPLIES
  FORALL(K:above(L)):
    kth_step(K)‘yy - kth_step(K-L)‘yy <=0.015
```

The proof is inductive on the number of steps K, using the same commands used for the theorem in the previous section

```
[-1] L * S <= 6/25
   |-------
 kth_step(K)‘yy - kth_step(K - L)‘yy <=  3/200
```

In the sequent, `[-1]` is the antecedent and `1` is the consequent. The property holds independently of the initial position of the robot on the line, only the constraint on the duration of the fault (`L * S`) must be satisfied. A similar theorem can be proved on the x coordinate.



$\mathbf{x}_1$
$\mathbf{x}_{N_i}$
Coordination FMU
$\mathbf{x}_d$
to neighbours
Control FMU
$\omega_1, \omega_2, \omega_3, \omega_4$
Plant FMU
$\phi, \theta, \psi, \dot{\phi}.\dot{\theta}, \dot{\psi}$
$\mathbf{x}, \dot{\mathbf{x}}$

Figure 6.7: Logical connections between the FMUs of a drone.

In particular, this theorem guarantees that if a fault changes the value of the left sensor to white for less than `L * S` time, then the fault will never move the robot from one side of the line to the other. This means that if the fault is detected within `L` steps, the robot is still close to the line. If the hypothesis of the theorem is not true, we do not have information. Figure 6.3b shows a co-simulation run for the previous fault with `L = 40` and `M = 80` and `S = 0.01`. Since $L * S = 0.4 > 0.24$, the hypothesis of the theorem is not met. In this case, the left sensor moves from the internal side of the path to the external side and the robot drives away from the painted line.

6.2 Cooperative UAVs

This section proposes a co-simulation architecture with three specialized interacting FMUs for each drone (Figure 6.7): a *plant* FMU for the drone dynamics, a *control* FMU to control direction and velocity, and a *coordination* FMU implementing the coordination protocol. The latter FMU receives information of the neighboring drones and computes the next desired position, which is fed to the control FMU and transmitted to the neighbors' coordination FMUs. The control FMU reads the feedback from the plant, consisting in the current position and the respective time derivatives and outputs the control commands to the plant FMU. The architecture of a co-simulation in a scenario of five drones is shown in Figure 6.8. All the FMUs are connected through the INTO-CPS co-simulation engine.

Generally, the rate of communication in the two cases above is different, as shown in Figure. 6.9. We distinguish between:

- the time discretization interval ϵ at which the target position of each drone is updated in the coordination algorithm, and
- a smaller step τ that is the numeric integration step used by the drone dynamics simulator for communications with the low-level controller of the drone.

From a logical point of view,

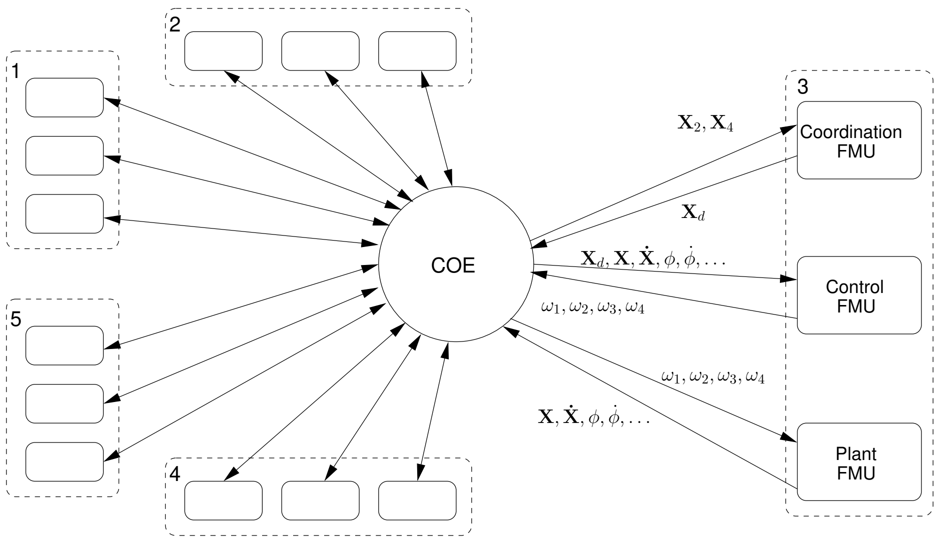

1
2
3
4
5
COE
$\mathbf{X}_2, \mathbf{X}_4$
$\mathbf{X}_d$
$\mathbf{X}_d, \mathbf{X}, \dot{\mathbf{X}}, \phi, \dot{\phi}, \ldots$
$\omega_1, \omega_2, \omega_3, \omega_4$
$\omega_1, \omega_2, \omega_3, \omega_4$
$\mathbf{X}, \dot{\mathbf{X}}, \phi, \dot{\phi}, \ldots$
Coordination FMU
Control FMU
Plant FMU

Figure 6.8: Co-simulation schema of a swarm of drones.

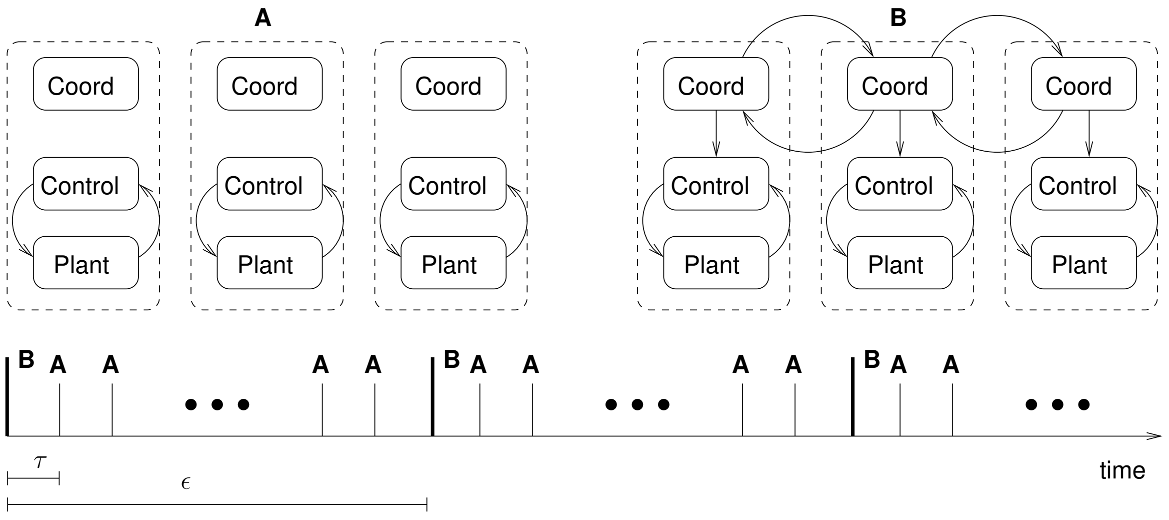


Figure 6.9: Data exchanges every τ sec. (A) and every ϵ sec. (B).

- τ is the interval at which the simulator for the drone dynamics (part of the plant FMU) and the simulator of the attitude/position controller (part of the control FMU) exchange data.

- ϵ is the interval at which the coordination FMUs of each drone compute the new target position and send it (i) to its immediate neighbors and (ii) to the Control FMU of the drone.

In Figure 6.9, events A and B show synchronization at τ and ϵ intervals, respectively.

As an application scenario, a variation of the classical formation control scheme, based on the well-known consensus protocol described in (Olfati-Saber et al., 2007), is considered. The algorithm in (Olfati-Saber et al., 2007) is distributed and allows drones to asymptotically converge to a target point.

The proposed coordination algorithm, instead, is obtained by the original one, simply assuming that two drones are fixed at the extreme of a line segment. This variant allows the uniform placement of the drones along the interval and it has not been studied in that work.

The proposed coordination algorithm is summarized with the following equations:

$$\begin{cases} x_1 &= \min \\ x_i(k+1) &= \epsilon x_{i-1}(k) + (1-2\epsilon)x_i(k) + \epsilon x_{i+1}(k), i \in [2 \,..\, N-1] \\ x_N &= \max\,. \end{cases} \tag{6.1}$$

where *min* and *max* are the extreme of the line segment, x_i is the position of the i-th drone along the line segment, k is the co-simulation step, and ϵ is the time discretization interval.

The framework has been applied in a scenario where a small number of quadcopters (5) are supposed to coordinate on the interval $[0, 100]$. The first and fifth drones are supposed to be stationary at the outer position of the interval, while the other three must recursively adjust their positions according to the shared coordination policy.

The rest of this section shows how the protocol can be specified in a PVS theory, validated through co-simulation, and convergence of the protocol can be analyzed by theorem proving.

PVS theory of the coordination protocol

Theory `coverage_state` (Listing 6.7) contains the type `location` which is defined as a single x coordinate, since the drones are assumed to be aligned. Fields `prec` and `foll` hold the location of the preceding and following drone, respectively. Field `timestep` is a counter that is reset every ϵ seconds, and counts the current number of co-simulation steps after the last reset. Its value ranges on the interval $[0..n]$ (*upto(n)*), where n, a parameter of the theory, equals ϵ/τ.

Theory `exec_coverage` (Listing 6.8) contains the function `exec_cvg` which is the PVS form of Equation. 6.1, where a, b, and c are the coordinates of the preceding, current, and following drone, respectively. Function `X` accesses the x-coordinate of the i-th drone in the system state. Function `exec_coverage` models the coverage protocol, updating the system map `drones`.

The `drones` field is an anonymous function (a λ-expression) of a drone identifier. This function extracts the coordinates of drone `i` and of its neighbors from the swarm state (using function `X`), and computes the new state of drone `i` according to the protocol.

```
coverage_state[N, n: posnat]: THEORY
BEGIN

drone_id: TYPE = below(N)

location: TYPE = [# x: real #]

state: TYPE+ = [#
                  prec: location,
                  self: location,
                  foll: location,
                  id: drone_id
                 #]

dmap: TYPE = [drone_id -> state]

%-- definition of system state
systemState: TYPE = [#
                      drones: dmap,
                      timesteps: upto(n)
                      #]

END coverage_state
```

Listing 6.7: Basics types of the protocol in PVS.

Function `tick` is needed to synchronize the different *Coordination* FMUs, as explained in Figure 6.9. In the `initial` theory (not shown), constant n is defined as the ratio *eps/stepsize*, i.e., ϵ/τ. Every n timesteps, the new desired position of the drone is computed by a step of the algorithm (`exec_coverge`).

Definitions for global parameters and for the initial swarm state are contained in the theory `initial`.

Co-simulation

The following subsections describe the different FMUs for the specific case study. The plant FMU is exported from OpenModelica, the controller FMU is a C program, and the coordination FMU is a PVS theory.

The plant FMU

The quadcopter's behavior as a function of the rotor speeds can be described by two linearized models, one for position:

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} g\phi \sin\psi_d + g\theta \cos\psi_d \\ -g\phi \cos\psi_d + g\theta \sin\psi_d \\ C_z(\omega_1 + \omega_2 + \omega_3 + \omega_4) \end{pmatrix} \tag{6.2}$$

```
exec_coverage: THEORY
BEGIN
IMPORTING initial, coverage_state[N]

exec_cvg(a,b,c: real, i: drone_id): real =
     IF (i = 0)
     THEN min_x
     ELSE
          IF (i = N - 1)
          THEN max_x
          ELSE eps*a + (1 - 2*eps)*b + eps*c
          ENDIF
     ENDIF

X(s: systemState, i: drone_id): real =
     s`drones(i)`self`x

exec_coverage(s: systemState) : systemState =
     s WITH
          [`drones :=
               LAMBDA (i: drone_id):
                    LET xp = if i > 0 then X(s, i - 1)
                              else 0 endif,
                         x = X(s, i),
                         xf = if i < N - 1 then X(s, i + 1)
                              else max_x endif
                    IN
                         s`drones(i) WITH [`self`x := exec_cvg(xp, x, xf, i)
    ]
          ]

tick(s: systemState) : systemState =
     IF (s`timesteps = n)
     THEN
          exec_coverage(s) WITH [timesteps := 0]
     ELSE
          s WITH [timesteps := s`timesteps + 1]
     ENDIF

kth_step(k: nat): RECURSIVE systemState =
     IF (k = 0) THEN
          initial
     ELSE
          tick(kth_step(k - 1))
     ENDIF
     MEASURE k

END exec_coverage
```

Listing 6.8: coordination algorithm in PVS.

and one for attitude:

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = \begin{pmatrix} C_\phi(\omega_2 - \omega_4) \\ C_\theta(\omega_3 - \omega_1) \\ C_\psi(\omega_1 - \omega_2 + \omega_3 - \omega_4) \end{pmatrix} . \tag{6.3}$$

In the above models, C_ϕ, C_θ, C_ψ, and C_z are physical constants of the quadcopter related to mass, rotational inertia, geometry, and rotor characteristics, g is the gravity acceleration, and ψ_d is the desired yaw angle.

The FMU is automatically generated from OpenModelica and allows the values of the parameters to be set before the beginning of the simulation.

The control FMU

For reasons of computational efficiency, control equations have been implemented in a C function wrapped in a 'hand-written' FMU. The position controller is defined by the following equation, where $e_x = x - x_d$ and $\mathrm{s}_d = (\sin \psi_d)/g$ etc. :

$$\begin{pmatrix} \phi_c \\ \theta_c \\ \omega_z \end{pmatrix} = \begin{pmatrix} -\mathrm{s}_d(2\lambda_P\dot{x} + \lambda_P^2 e_x) + \mathrm{c}_d(2\lambda_P\dot{y} + \lambda_P^2 e_y) \\ -\mathrm{c}_d(2\lambda_P\dot{x} + \lambda_P^2 e_x) - \mathrm{s}_d(2\lambda_P\dot{y} + \lambda_P^2 e_y) \\ -\frac{2\lambda_P}{C_z}\dot{z} + \frac{\lambda_P^2}{C_z}e_z) \end{pmatrix} , \tag{6.4}$$

and the attitude controller is defined by

$$\begin{pmatrix} \omega_\phi \\ \omega_\theta \\ \omega_\psi \end{pmatrix} = \begin{pmatrix} -\frac{2\lambda_A}{C_\phi}\dot{\phi} - \frac{\lambda_A^2}{C_\phi}(\phi - \phi_c)) \\ -\frac{2\lambda_A}{C_\theta}\dot{\theta} - \frac{\lambda_A^2}{C_\theta}(\theta - \theta_c)) \\ -\frac{2\lambda_A}{C_\psi}\dot{\psi} - \frac{\lambda_A^2}{C_\psi}(\psi - \psi_d)) \end{pmatrix} \tag{6.5}$$

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix} = \begin{pmatrix} \frac{\omega_z}{4} - \frac{\omega_\theta}{2} + \frac{\omega_\psi}{4} \\ \frac{\omega_z}{4} + \frac{\omega_\phi}{2} - \frac{\omega_\psi}{4} \\ \frac{\omega_z}{4} + \frac{\omega_\theta}{2} + \frac{\omega_\psi}{4} \\ \frac{\omega_z}{4} - \frac{\omega_\phi}{2} - \frac{\omega_\psi}{4} \end{pmatrix} , \tag{6.6}$$

where λ_p and λ_a are design parameters of the controller.

The coordination FMU

The coordination FMU is generated using the framework proposed in this thesis; it embeds the PVS theory in Listing 6.10.

Theories `fmu_coverage_state` (Listing 6.9 and theory `fmu_exec_coverage` (Listing 6.10) are a simplification of `coverage_state` and `exec_coverage`. Equation 6.1 is expressed by function `fmu_exec_cvg` with the same algorithm as function `exec_-cvg` above, whereas `fmu_tick` merges functions `exec_coverage` and `tick` above. It

```
fmu_coverage_state[N: posnat]: THEORY
BEGIN

drone_id: TYPE = below(N)

location: TYPE = real

state: TYPE+ = [#
                  prec: location,
                  self: location,
                  foll: location,
                  id: drone_id
                #]

simstate: TYPE = [#
                     timesteps: integer,
                     st: state
                   #]

END fmu_coverage_state
```

Listing 6.9: State for the coordination algorithm.

can be easily proved that `fmu_exec_cvg` is equivalent to `exec_coverage`. Therefore the properties that will be verified on `exec_coverage` also apply to `fmu_exec_-coverage` and the validation performed using `fmu_exec_coverage` also applies to `exec_coverage`.

A co-simulation run

Co-simulation is executed in the scenario reported in Figure 6.10a. The graphic rendering of the co-simulation created with PVSio-web.

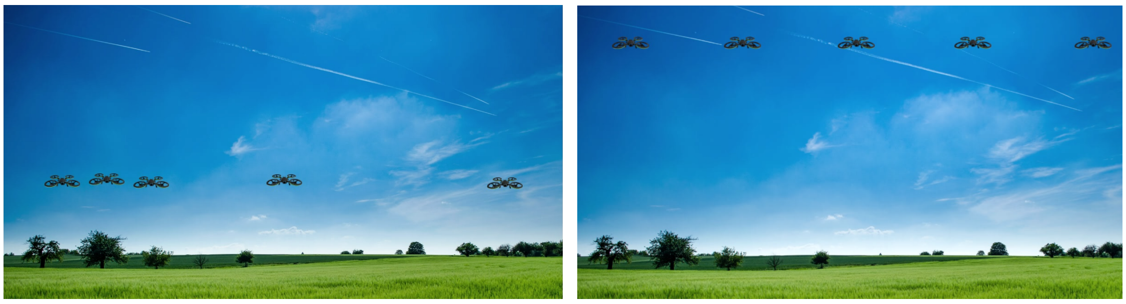

(a) Begin of a co-simulation run. (b) End of a co-simulation run.

Figure 6.10: Graphic rendering of the co-simulation.

Assuming as a reference the leftmost UAV in position 0, in the initial deployment the second, third and fourth drones, are, respectively, placed 10, 20, and 50 meters after the first one. The last UAV is placed at the last extreme of the line segment, i.e. 100 meters after the first drone.

```
fmu_exec_coverage: THEORY
BEGIN
IMPORTING fmu_initial,fmu_coverage_state[N]

fmu_exec_cvg(a, b, c: real, i: drone_id): real =
     IF (i = 0)
     THEN min
     ELSE
         IF (i = N - 1)
         THEN max
         ELSE eps*a + (1 - 2*eps)*b + eps*c
         ENDIF
     ENDIF

fmu_tick(s: simstate) : simstate =
     LET xp = if (s`st`id > 0) then s`st`prec
              else min endif,
         x = s`st`self,
         xf = if (s`st`id < N - 1) then s`st`foll
              else max endif
     IN
     COND
     s`timesteps >= eps/stepsize ->
         s WITH [`st`self := fmu_exec_cvg(xp, x, xf, s`st`id),
                 `timesteps := 0],
     s`timesteps < eps/stepsize ->
         s WITH [`timesteps := s`timesteps + 1]
     ENDCOND



END fmu_x_coverage
```

Listing 6.10: Executable theory for the coordination algorithm.

The co-simulation is run with a fixed stepsize of 0.05, assuming $\epsilon = 1/4$. Each mobile drone will start to communicate with the adjacent UAV and to make some position change, according to the formula shown above. After 10 simulated seconds the coordination algorithm has reached the reference points (25, 50, 75 for the moving drones with ID 2,3,4 respectively). Figure 6.10a shows the position of the drones at the beginning of the simulation while Figure 6.10b shows the final positions.

Figure 6.11 shows in details the evolution of the x coordinate for each drone over time. Co-simulation is used to validate the protocol. For example, the same co-simulation shown above with $\epsilon = 3/4$ has a different result, shown in Figure 6.12, because the drones oscillate and the algorithm does not converge.

Formal Verification

This section presents results on the formal verification using the abstract theory `exec_coverage`. Various properties are proved, the main one being the *convergence* of the drones to their target positions. Fundamental in the proof of convergence, are properties of the sum of positions of the drones at each step and the constraint on ϵ.

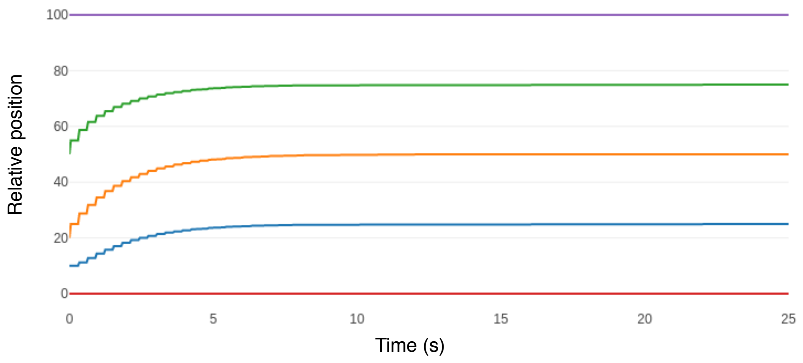


Figure 6.11: Plot of the desired x of the drones applying the algorithm with $\epsilon = 1/4$.

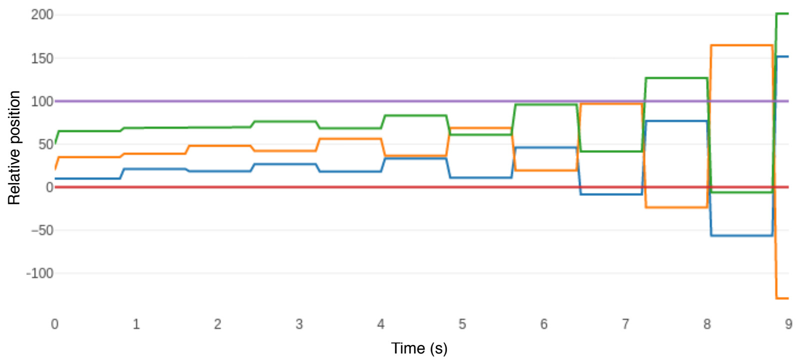


Figure 6.12: Plot of the desired x of the drones applying the algorithm with $\epsilon = 3/4$.

Definitions and assumptions

For simplicity, it is assumed that the segment to be spanned by the drones starts at the origin of the x axis, i.e., $min = 0$ and $max = d$, where d the length of the segment. In the following, h, i, j, and k denote natural numbers.

The following definitions will be used:

1. $N \geq 3$ is the total number of drones, numbered starting from 0;
2. $L = d/(N-1)$ is the desired distance between adjacent drones;
3. $\bar{X}_i = iL$ is the target position of drone i;
4. X_i^k is the position of drone i at step k;
5. $X^k = (X_0^k, \cdots, X_{N-1}^k)$ is the placement of the drones at step k;
6. the initial placement is X^0;
7. $\epsilon \in [0,1]$ is the step size;

8. $c^k = \sum_{i=0}^{N-1} X_i^k$ is the sum of the drone coordinates at step k.

With the above definitions, we assume that:

1. the drones are numbered in order of increasing distance from X_1^0, i.e., $\forall_{i<N-1} X_i^0 < X_{i+1}^0$; the drones with index $i \in [1 \mathrel{..} N-2]$ will be called *internal*;
2. The first and last drones are initially positioned at $X_0^0 = 0$ and $X_{N-1}^0 = d$, respectively;
3. initially, each internal drone is placed before its target position: $\forall_{i \in [1 \mathrel{..} N-2]} X_i^0 < \bar{X}_i$;

With all the above assumptions it can be proved that:

Theorem 1 (no_cross). *If* $\epsilon < 1/3$*, then* $\forall_{k \geq 0} \forall_{i \in [0 \mathrel{..} N-2]} X_i^k < X_{i+1}^k$.

the drones maintain their relative spatial ordering, i.e., their trajectories do not cross, with the above assumptions and a constraint on ϵ*.*

The Theorem can be expressed in PVS as follows (the assumption becomes an `AXIOM`):

```
init_ax: AXIOM
  FORALL (i:subrange(N-1)): initial(i) < initial(i+1) ;

no_cross: THEOREM
  eps <= 1/3 IMPLIES
   FORALL (k: nat):
     FORALL (i:{ n:nat | n < N-1}):
        kth_step(k)(i) < kth_step(k)(i+1)
```

Appendix A shows details on the proof strategy used for the Theorem.

Theorem 2 (reference position upper bound). *If* $\epsilon < 1/2$*, then* $\forall_{k \in \mathbb{N}} \forall_{i \in [1 \mathrel{..} N-2]} X_i^k \leq iL$.

the target position of each drone is an upper bound for the position at each step.

Theorem 3 (non decreasing). *(a) If* $\epsilon < 1/2$*, then* $\forall_k c^{k+1} \geq c^k$*; (b)* $\forall_i X_i^k = iL \implies c^{k+1} = c^k$.

the sums of the drone coordinates at each step are a non-decreasing sequence, and if all drones are at their target position, then $c^{k+1} = c^k$*.*

Theorem 4 (convergence). *Let $\epsilon < 1/3$ and $\forall_{i \in [0\,..\,N-2]} X_i^0 < L$. We have that:*
(a) $\forall_k((\forall_i X_i^k = iL) \implies \forall_{h \geq 1} c^{k+h} = c^k)$, and (b) $\forall_k(\forall_{h \geq 1} c^{k+h} = c^k) \implies \forall_i X_i^k = iL)$.

there is a relationship between the sequence in time (i.e., with respect to simulation steps) of the sums of drone coordinates and the drone positions at each step. This relationship is used in further theorems.

Theorem 5 (globally increasing). *If there is a step* k *such that $c^{k+1} = c^k$ and one or more drones are not at the target position, then there is a number $h > 1$ such that $c^{k+h+1} > c^{k+h}$.*

$$[\exists_k(c^{k+1} = c^k \wedge \exists_{j \geq 3}(X_j^k < iL \wedge \forall_{i<j} X_i^k = iL))] \implies \exists_{h \geq 1} c^{k+h+1} > c^{k+h}$$

the sum of the drone coordinates may remain constant for some number of steps, but will eventually increase unless all drones have reached their target position.

Theorem 6 (limit). $\forall_{i \in [0\,..\,N-1], k \in \mathbb{N}} \lim_{k \to \infty} X_i^k = iL.$

the position of each drone approaches monotonically the drone's target position as k increases.

The convergence of the algorithm cannot be proved with the same approach used in (Olfati-Saber et al., 2007), because the required assumptions on the eigenvalues of the Jacobian do not apply on the modified version of the algorithm used in this case study.

6.3 Integrated Clinical Environment

The considered system is an Integrated Clinical Environment (ICE) system for intensive care of patients (Goldman, 2008). It includes three medical devices:

- an infusion pump injecting a pain remedy drug in the bloodstream of the patient;
- a monitor for checking the vital signs of the patient;
- a supervisor device running a safety interlock application that automatically stops the pump when the monitor detected parameters are out of range.

Clinicians can set up therapy parameters and monitor the patient's condition by interacting with the user interfaces of the devices. The visual appearance of the medical devices is shown in Figure 6.13. A description of the user interface and functionalities of the three devices is now provided.

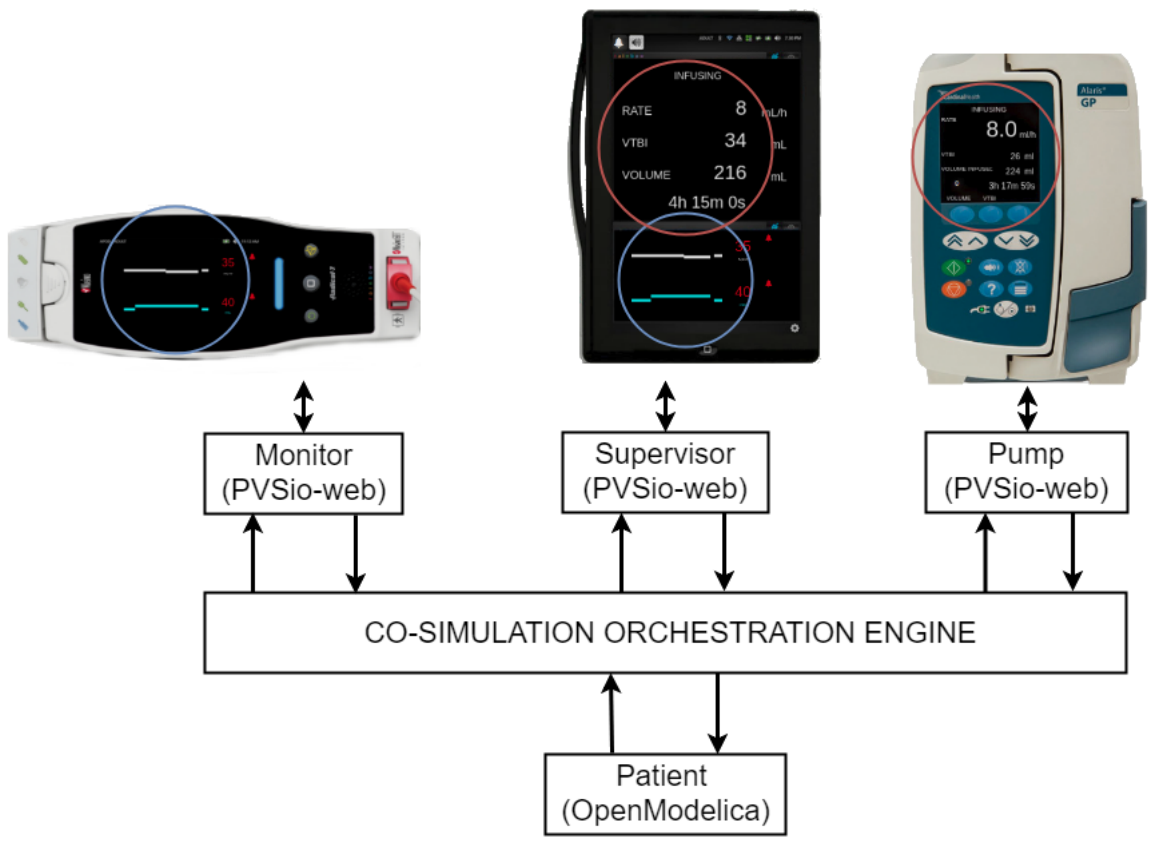


Figure 6.13: Co-Simulation architecture of ICE case study.

- The front panel of the patient monitor shows two vital signs: oxygen saturation level (SpO_2) in the upper half of the monitor display, and heart rate (HR) in the lower half of the monitor display. The current value of a vital sign is reported using a numeric display. A trace display shows the temporal evolution for the last 25 seconds of the monitored vital signs. Each monitored parameter has safe range limits. The monitor triggers an alarm if these limits are exceeded.

- The front panel of the pump provides a display and a number of buttons that can be used to enter the Volume To Be Infused (VTBI) and the infusion rate, as well as to start/stop the infusion. When the infusion is running, the display of the pump shows the infusion rate, the VTBI, the volume already infused, and the time to complete the infusion.

- The supervisor device provides a front-panel that can be used for remote monitoring of the pump and patient monitor. It is a portable device with a display divided into two sections. The upper section mirrors the pump display, and the lower section mirrors the monitor display.

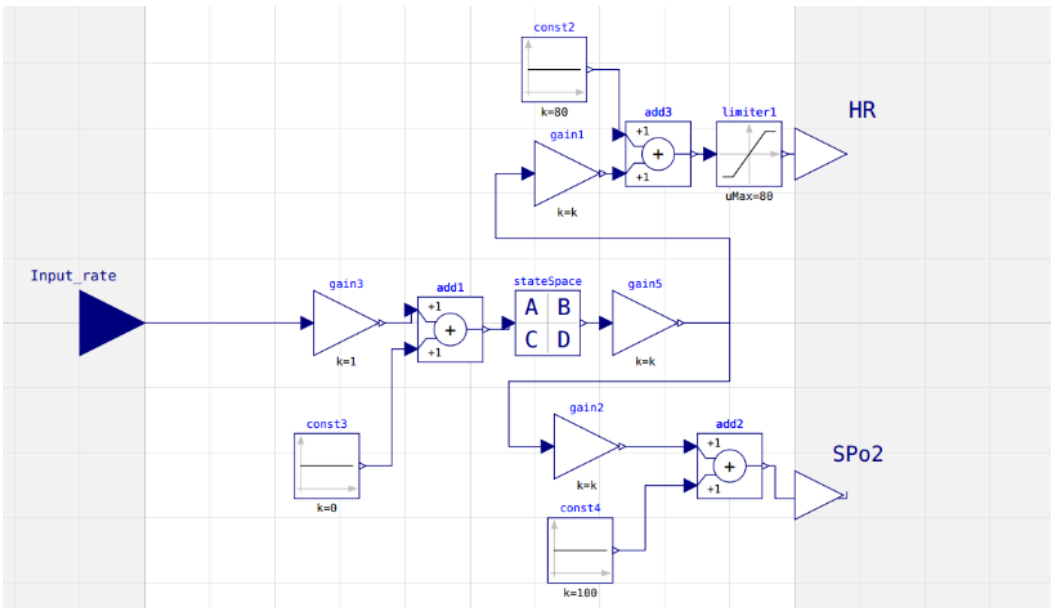
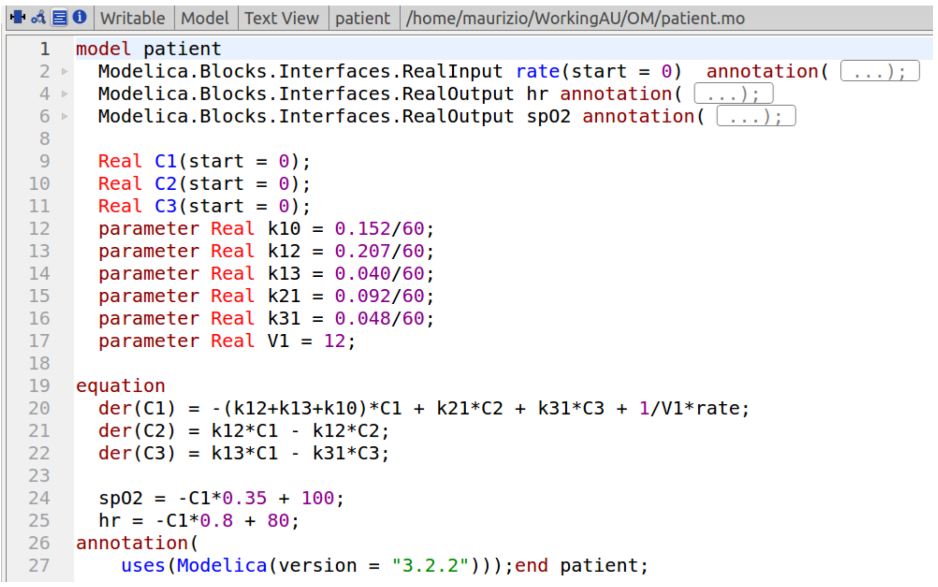


Figure 6.14: Patient model in OpenModelica (block diagram).


Writable | Model | Text View | patient | /home/maurizio/WorkingAU/OM/patient.mo

```
model patient
  Modelica.Blocks.Interfaces.RealInput rate(start = 0)  annotation( ...);
  Modelica.Blocks.Interfaces.RealOutput hr annotation( ...);
  Modelica.Blocks.Interfaces.RealOutput sp02 annotation( ...);

  Real C1(start = 0);
  Real C2(start = 0);
  Real C3(start = 0);
  parameter Real k10 = 0.152/60;
  parameter Real k12 = 0.207/60;
  parameter Real k13 = 0.040/60;
  parameter Real k21 = 0.092/60;
  parameter Real k31 = 0.048/60;
  parameter Real V1 = 12;

equation
  der(C1) = -(k12+k13+k10)*C1 + k21*C2 + k31*C3 + 1/V1*rate;
  der(C2) = k12*C1 - k12*C2;
  der(C3) = k13*C1 - k31*C3;

  sp02 = -C1*0.35 + 100;
  hr = -C1*0.8 + 80;
annotation(
    uses(Modelica(version = "3.2.2")));end patient;
```

Figure 6.15: Differential equations used in the OpenModelica model.

The devices are modeled in PVS. The models were developed in previous work (Masci et al., 2015c; Harrison et al., 2017) by reverse engineering real medical devices.

The patient model is developed in this thesis in OpenModelica. It is based on a pharmacokinetic model (Bequette, 2003) describing how the human body absorbs anaesthetic drugs injected intravenously. The model uses 3-compartments to represent the changes of drug concentration in plasma (first compartment), highly per-

```
fmi_module.create_FMU("Monitor",{
  fmi: [{ name:"sp02", type:"Real", variability:"discrete",
          scope:"input", value:"0" },
        { name:"HR", type:"Real", variability:"discrete",
          scope:"input", value:"0" },
        { name:"sp02_output", type:"Real", variability:"discrete",
          scope:"output", value:"0" },
        { name:"HR_output", type:"Real", variability:"discrete",
          scope:"output", value:"0" },
        { name:"isOn", type:"Boolean", variability:"discrete",
          scope:"output", value:"0" }],
  init: "init",
  tick: "tick" });
```

Listing 6.11: Generation of the FMU of the patient monitor.

fused tissues (second compartment) and scarcely perfused tissues (third compartment).

The structure of the developed OpenModelica model is shown in Figure 6.14. The equations of the 3-compartment model are embedded in the `stateSpace` block. Figure 6.15 shows the differential equations of the kinetic behavior of the 3-compartment scheme in OpenModelica textual language. The values of the parameters are taken from (Bequette, 2003). The structure of the model was inspired by the Simulink model developed in (Pajic et al., 2014).

The FMUs for the device models are automatically generated using the developed framework. Listing 6.11 shows the specific invocation of the framework's APIs for generating the FMU for the patient monitor. The FMU of the patient model is generated using OpenModelica.

Co-simulation run

Co-simulation runs can be executed to validate the behavior of the supervisor, e.g., to check that, during an infusion, the supervisor device automatically stops the infusion if the patient monitor signals that both monitored vital signs have abnormal values.

Figure 6.16 shows a diagram of the monitored vital signs for a co-simulation run, as well as the value of the infusion rate at the bottom of the graph. When the infusion is started, the values of HR and SpO_2 start to decrease.

The threshold for triggering an alarm for HR is 60 beats per minute, while the threshold for $Sp0_2$ is 88%. In the simulated scenario, both vital signs exceed the respective thresholds after 25 minutes (1,500 seconds in Figure 6.16).

At this point, the supervisor stops the infusion — the value of the infusion rate in the graph becomes 0. When the infusion is paused, the values of HR and $Sp0_2$ start to get back to a normal range.

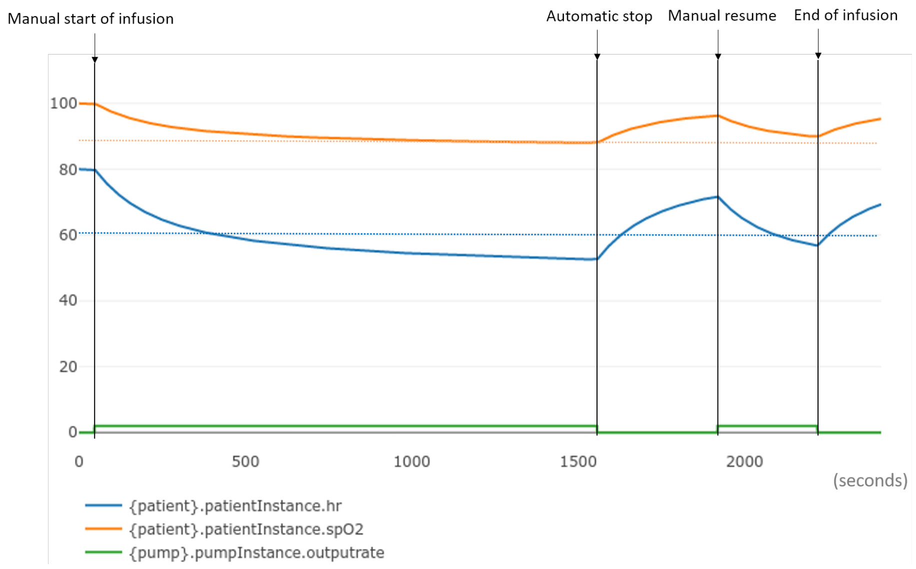


Figure 6.16: Vital signs and infusion rate in a co-simulation run.

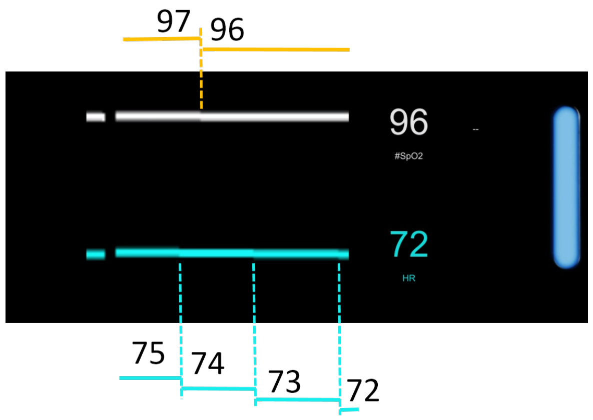


Figure 6.17: Monitor display provides little insight if trace duration is 25 seconds.

The introduction of the patient model in the co-simulation made it possible to gain useful insights on the device models. For example, it is interesting to note that the setup of the display of the patient monitor, including range and temporal duration of the trace, needs to be carefully evaluated, otherwise the temporal evolution of the vital signs on the display provides little information about what is happening.

The initial setup used in the co-simulation adopted a temporal duration of 25 seconds for the trace. As a result, the trace was always showing a flat line with few variations (see Figure 6.17) rather than the diagram shown in Figure 6.16.

These information can be fed in the model, e.g., by adding constraints on the parameters that can be used to setup the device. This will help formal methods expert make sure that requirements of the device are verified under correct hypotheses

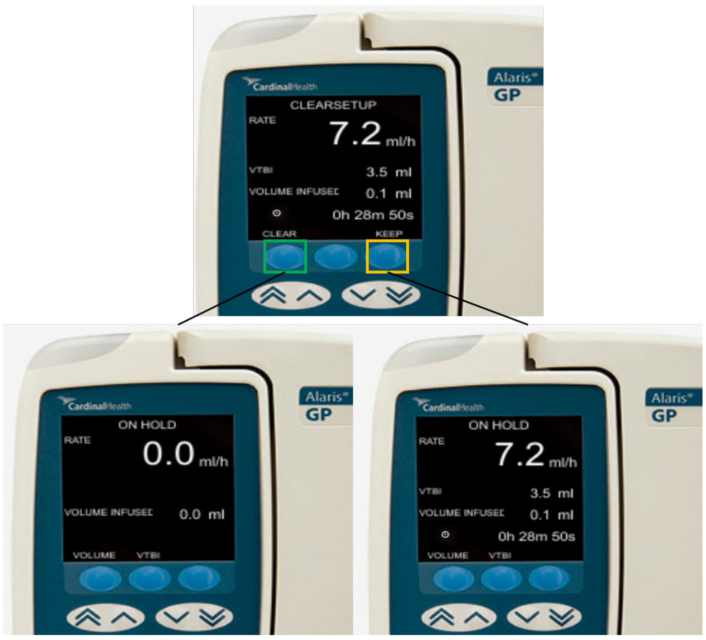


Figure 6.18: Displays shown by the pump when clearing settings. Buttons pressed necessary to perform the clear settings sequence are highlighted in the pictures.

about the device setup.

Formal verification

In this section it is shown that PVS allows to check a use-related safety requirements.

The considered requirement is taken from a set of generic safety requirements identified by the Food and Drug Administration (FDA) for infusion pumps (Arney et al., 2012). This requirement is specific of the infusion pump. It aims to mitigate the risk of inadvertently changing the settings of the infusion pump as a consequence of accidental button presses performed on the front-panel of the pump. The requirement reads as follows:

> **R1**: *Clearing the pump settings shall require confirmation.*

The considered requirement contains two main concepts: *clearing settings* and *require confirmation*. In the pump under analysis, clearing settings can be performed by turning the pump off and then on. After this action, settings are not cleared yet — the pump shows a confirmation screen where the user needs to confirm the operation. The confirmation screen is characterized by a top line display showing a query to the user (*clear setup?*). In this screen, two actions are provided, one to confirm the operation, the other to abort.

Figure 6.18 shows the confirmation screen presented by the pump after turning the pump off and on. The string `CLEAR SETUP` shown at the top of the screen is the query. The labels on the buttons at the bottom of the screen indicate the functionalities of the buttons. Pump settings are cleared if the the button of the left (*clear*) is pressed. Settings are not cleared if the other button (*keep*) is pressed.

These functionalities are captured in the PVS model of the pump. Formal methods experts can use the PVSio-web prototype to demonstrate these functionalities of the PVS model to other team members.

Demonstrations performed with the PVSio-web prototype allows to check the key sequence for specific scenarios. The PVS theorem prover can be used to extend these scenario-based analyses to all possible scenarios. That is, the theorem prover can be used to check that, in all pump states, if the clear settings sequence is performed, the pump will always ask confirmation to the user before clearing the settings.

Two expressions are created to specify the identified concepts in terms of actions and states of the pump specification:

```
clear_settings(st: alaris): alaris = power_on(power_off(st))

require_confirmation(st: alaris): bool = st‘topline = clearsetup
```

The first expression (`clear_settings`) is a transition function that takes the current device state as argument and returns a new devices state. It uses the composition of two actions, `power_on` and `power_off` defined in the PVS specification of the pump, to turn off and then on the pump.

The second expression (`require_confirmation`) is a function that returns a boolean type. The function checks whether the state attribute representing the top line display of the pump is presenting a query to the user. In the developed model, this is indicated by an enumerated constant `clearsetup`.

The following theorem uses the expressions defined above to check whether the pump requires confirmation whenever the user tries to clear settings:

```
R1: THEOREM
 FORALL (st: alaris):
   require_confirmation(clear_settings(st))
```

In the formalization, the universal quantifier `FORALL` is used to indicate that the requirement should hold in all possible states.

For the analysis of theorem R1, the *grind* rule can be conveniently used to complete the proof. It performs automatic expansion of terms and heuristic instantiation of definitions. Using this rule on the initial formulation of R1, the theorem prover finds a counter-example.

The counter-example indicates that the pump does not require confirmation when the current value of VTBI, infusion rate, and time are already 0.

Formal methods experts can use the PVSio-web prototype to demonstrate this counter-example to other team members. A discussion with domain specialists is in fact usually necessary at this point to understand whether this is a concern, or whether this case can be safely discarded. In this case, it can be argued that this is not a concern — it is not necessary to ask confirmation to reset a value when the value is already 0.

The initial formulation of the theorem is therefore refined to exclude the identified case:

```
   R1v1: THEOREM
     FORALL (st: (powered_on?)):
         (device(st)‘vtbi /= 0 OR
           device(st)‘infusionrate /= 0 OR
           device(st)‘time /= 0)
     IMPLIES
     require_confirmation(clear_settings(st))

clear_settings(st: alaris): alaris =
        power_on(power_off(st))
require_confirmation(st: alaris): bool =
        st‘topline = clearsetup
```

In the new formulation, *predicate subtyping* is used to narrow down the set of states to those where the pump is powered on — the counter-example returned by the theorem prover indicates also that the sequence for clearing settings can be initiated only when the device is powered on. A predicate subtype is used to add this condition in the theorem. This is indicated by adorning the predicate name with round brackets. The conditions for VTBI, infusion rate and time are introduced in the theorem using disjunction and implication (in the PVS syntax, the symbol for inequality is `/=`).

Re-running the proof on theorem `R1v1` succeeds, indicating that the pump design complies with this formulation of the requirement.

Chapter 7

Conclusions

This thesis proposes an approach and a framework for model-based design of Cyber-Physical Systems (CPS) based on co-simulation and formal verification. A template is shown for developing logic theories in the formal language of the Prototype Verification System (PVS) that can be integrated as a standard element in a co-simulation framework and formally analyzed by the theorem prover of PVS. Co-simulation allows validation of the system behavior, theorem prover technology allows the generalization of properties to all the states of the system.

The developed framework has been applied to some case studies from different application domains. The case studies demonstrate how the co-simulation may provide useful information for the verification process. One of the advantages of using PVS as modeling tool, is the possibility of generating interactive prototypes: the developed framework supports the automatic creation of PVS-based FMU with a Graphical User Interface.

These integrated simulations provide formal methods experts with a convenient means to explain formal models and formal analysis results to a multi-disciplinary team of developers that may not be familiar with formal methods. This engagement supports validation activities necessary to check whether formal models correctly capture the intended behavior of the system, and to discuss formal analysis results. This has been demonstrated using a realistic case study in the medical domain.

Appendix A

An example of PVS Proof

Excerpts from the proof trace of Theorem 1 (see Section 6.2) are shown in this section. The first step is a simple rearrangement (*flattening*) of the goal:

```
no_cross :
   |-------
{1}   eps <= 1 / 3 IMPLIES
        (FORALL (k: nat):
           FORALL (i: {n: nat | n < N - 1}):
             X(kth_step(k), i) < X(kth_step(k), i + 1))

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
no_cross :
{-1}  eps <= 1 / 3
   |-------
{1}   FORALL (k: nat):
         FORALL (i: {n: nat | n < N - 1}):
           X(kth_step(k), i) < X(kth_step(k), i + 1)
```

Then, the *induct* command causes the theorem prover to start a proof by induction on *k*, generating the base case *no_cross.1* and the induction step *no_cross.2*:

```
Rule? (induct k)
Inducting on k on formula 1,
this yields  2 subgoals:
no_cross.1 :
[-1]  eps <= 1 / 3
   |-------
{1}   FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(0), i) < X(kth_step(0), i + 1)
```

The base case is proved in few steps (not shown) using two axioms (not shown) corresponding to Definition 6 and Assumption 1, added to the antecedents with the *lemma*.

The induction step is:

```
no_cross.2 :
[-1]  eps <= 1 / 3
   |-------
{1}   FORALL j:
         (FORALL (i: {n: nat | n < N - 1}):
            X(kth_step(j), i) < X(kth_step(j), i + 1))
          IMPLIES
          (FORALL (i: {n: nat | n < N - 1}):
             X(kth_step(j + 1), i) < X(kth_step(j + 1), i + 1))
```

The outermost quantifier is eliminated by the *skolem* command, which replaces the occurrences of the induction variable *j* in formula {1} with the new constant *j*, which "happens" to have the same name of the variable it replaces (another name could have been chosen, or generated by the theorem prover):

```
Rule? (skolem 1 j)
For the top quantifier in 1, we introduce Skolem constants: j,
this simplifies to:
no_cross.2 :
[-1]  eps <= 1 / 3
   |-------
{1}   (FORALL (i: {n: nat | n < N - 1}):
          X(kth_step(j), i) < X(kth_step(j), i + 1))
        IMPLIES
        (FORALL (i: {n: nat | n < N - 1}):
           X(kth_step(j + 1), i) < X(kth_step(j + 1), i + 1))
```

After flattening and skolemizing again, we obtain:

```
no_cross.2 :
[-1]  FORALL (i: {n: nat | n < N - 1}):
        X(kth_step(j), i) < X(kth_step(j), i + 1)
[-2]  eps <= 1 / 3
   |-------
{1}   X(kth_step(j + 1), i) < X(kth_step(j + 1), i + 1)
```

where *j* is a constant in formulas [-1] and {1}, while *i* is a variable in [-1] and a constant in {1}. The definition of *kth_step* is expanded in {1}, and then the one of *tick*:

then the

```
Rule? (rewrite kth_step 1)
...
Rule? (expand tick 1 1)
Expanding the definition of tick,
this simplifies to:
no_cross.2 :
[-1]  FORALL (i: {n: nat | n < N - 1}):
        X(kth_step(j), i) < X(kth_step(j), i + 1)
[-2]  eps <= 1 / 3
   |-------
  {1}   X(IF (kth_step(j)‘timesteps = n)
           THEN exec_coverage(kth_step(j)) WITH [timesteps := 0]
        ELSE kth_step(j) WITH [timesteps := 1 + kth_step(j)‘timesteps]
        ENDIF,
        i)
       < X(tick(kth_step(j)), 1 + i)
```

The first argument of *X* is a conditional expression. The *lift-if* command pushes the conditional to the outermost level:

```
Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
no_cross.2 :
[-1]  FORALL (i: {n: nat | n < N - 1}):
        X(kth_step(j), i) < X(kth_step(j), i + 1)
[-2]  eps <= 1 / 3
   |-------
{1}   IF (kth_step(j)`timesteps = n)
        THEN X(exec_coverage(kth_step(j)) WITH [timesteps := 0], i) <
              X(tick(kth_step(j)), 1 + i)
       ELSE X(kth_step(j) WITH [timesteps := 1 + kth_step(j)`timesteps],
i)
              < X(tick(kth_step(j)), 1 + i)
       ENDIF
```

The *split* command expands the conditional in two complementary implications, i.e.,

```
    (kth_step(j)`timesteps = n) IMPLIES
     X(exec_coverage(kth_step(j)) WITH [timesteps := 0], i) <
      X(tick(kth_step(j)), 1 + i)
```

and

```
    NOT (kth_step(j)`timesteps = n) IMPLIES
     X(kth_step(j) WITH [timesteps := 1 + kth_step(j)`timesteps], i) <
      X(tick(kth_step(j)), 1 + i)
```

thus yielding two subgoals. The first one has the more complex subproof, since it represents the case when the system state is updated:
assert

```
no_cross.2.1 :
[-1]  FORALL (i: {n: nat | n < N - 1}):
        X(kth_step(j), i) < X(kth_step(j), i + 1)
[-2]  eps <= 1 / 3
   |-------
{1}   (kth_step(j)`timesteps = n) IMPLIES
       X(exec_coverage(kth_step(j)) WITH [timesteps := 0], i) <
        X(tick(kth_step(j)), 1 + i)
```

The complexity of the proof is mainly due to the structure of the functions involved, but the proof strategy is quite simple. First, the sequent is reduced to the form

```
no_cross.2.1
[-1]  (kth_step(j)‘timesteps = n)
[-2]  FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(j), i) < X(kth_step(j), i + 1)
[-3]  eps <= 1 / 3
   |-------
{1}   X(exec_coverage(kth_step(j)), i) < X(tick(kth_step(j)), 1 + i)
```

Then, the inequality in {1} is transformed by repeated expansions of *x_coverage* and *tick*. This produces many branches due to the conditional expressions in the functions. In each branch, the resulting inequalities are solved by instantiating the induction hypothesis [1] and doing algebraic manipulations. For example, the following sequent:

```
no_cross.2.1.1.1.1.1.1 :
{-1}  1 + i < N - 1
[-2]  i > 0
[-3]  (kth_step(j)‘timesteps = n)
[-4]  FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(j), i) < X(kth_step(j), i + 1)
[-5]  eps <= 1 / 3
   |-------
  {1}  exec_cvg(X(kth_step(j), i - 1), X(kth_step(j), i),
              X(kth_step(j), 1 + i), i)
        <
        exec_cvg(X(kth_step(j), i), X(kth_step(j), 1 + i),
              X(kth_step(j), 2 + i), 1 + i)
```

is on the branch corresponding to the conditions *kth_step(j)‘timesteps = n*, $i > 0$, and $i < N - 2$. Expanding *exec_cvg* yields, with a couple of transformations,

```
no_cross.2.1.1.1.1.1.1 :
[-1]  1 + i < N - 1
[-2]  i > 0
[-3]  (kth_step(j)‘timesteps = n)
[-4]  FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(j), i) < X(kth_step(j), 1 + i)
[-5]  eps <= 1/3
   |-------
  {1}   X(kth_step(j), i) - 2 * (X(kth_step(j), i) * eps) +
        eps * X(kth_step(j), i - 1)
        + eps * X(kth_step(j), 1 + i)
        <
        X(kth_step(j), 1 + i) - 2 * (X(kth_step(j), 1 + i) * eps) +
         eps * X(kth_step(j), 2 + i)
         + eps * X(kth_step(j), i)
```

The induction hypothesis {4} is then instantiated successively with $i-1$, i, and $i+1$, obtaining a sequent with the three inequalities [-5], [-6], and [-7] in the antecedent:

```
no_cross.2.1.1.1.1.1.1 :
[-1]  1 + i < N - 1
[-2]  i > 0
[-3]  (kth_step(j)‘timesteps = n)
[-4]  FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(j), i) < X(kth_step(j), 1 + i)
{-5}  X(kth_step(j), i + 1) < X(kth_step(j), 1 + (i + 1))
[-6]  X(kth_step(j), i) < X(kth_step(j), 1 + i)
[-7]  X(kth_step(j), i - 1) < X(kth_step(j), 1 + (i - 1))
[-8]  eps <= 1/3
   |-------
  {1}   X(kth_step(j), i) - 2 * (X(kth_step(j), i) * eps) +
        eps * X(kth_step(j), i - 1)
        + eps * X(kth_step(j), 1 + i)
        <
        X(kth_step(j), 1 + i) - 2 * (X(kth_step(j), 1 + i) * eps) +
         eps * X(kth_step(j), 2 + i)
         + eps * X(kth_step(j), i)
```

Further manipulations and some utility lemmas (not shown) make it possible to assemble an antecedent formula matching the consequent, thus solving the subgoal:

```
no_cross.2.1.1.1.1.1.1.1.1.1.1.1 :
[-1]  X(kth_step(j), i) - 3 * eps * X(kth_step(j), i) <
        X(kth_step(j), 1 + i) - 3 * eps * X(kth_step(j), 1 + i)
[-2]  X(kth_step(j), i - 1) * eps < X(kth_step(j), 2 + i) * eps
[-3]  1 + i < N - 1
[-4]  i > 0
[-5]  (kth_step(j)`timesteps = n)
[-6]  FORALL (i: {n: nat | n < N - 1}):
         X(kth_step(j), i) < X(kth_step(j), 1 + i)
[-7]  X(kth_step(j), 1 + i) < X(kth_step(j), 2 + i)
[-8]  X(kth_step(j), i) < X(kth_step(j), 1 + i)
[-9]  X(kth_step(j), i - 1) < X(kth_step(j), i)
[-10] eps <= 1/3
   |-------
  {1}   ((X(kth_step(j), i - 1)) * eps) + (X(kth_step(j), i)) -
        3 * ((X(kth_step(j), i)) * eps)
        <
        ((X(kth_step(j), 2 + i)) * eps) + (X(kth_step(j), 1 + i)) -
         3 * ((X(kth_step(j), 1 + i)) * eps)

Rule? (add-formulas -1 -2)
Adding formulas -1 and -2,

This completes the proof of no_cross.2.1.1.1.1.1.1.1.1.1.1.1.
```

Appendix B

Publications

Peer reviewed workshop papers

1. **M. Palmieri**, C. Bernardeschi, P. Masci, "Co-simulation of semi-autonomous systems: the line follower robot case study", *International Conference on Software Engineering and Formal Methods*, pages:423–437, 2017. **Candidate's contributions**: Designed and implemented the co-simulation interface, inspected the case study, run co-simulation experiments.

2. **M. Palmieri**, C. Bernardeschi, A. Domenici, A. Fagiolini, "Co-simulation of UAVs with INTO-CPS and PVSio-web", *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages:52–57, 2018. **Candidate's contributions**: Implemented the model of the UAV, created the graphic interface and run co-simulation experiments.

3. **M. Palmieri**, C. Bernardeschi, P. Masci, "A Flexible Framework for FMI-Based Co-Simulation of Human-Centred Cyber-Physical Systems", *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages:21–33, 2018. **Candidate's contributions**: Designed and implemented the framework, used the framework to create the case study, run co-simulations experiments and analyzed the outputs.

4. A. Domenici, A. Fagiolini, **M. Palmieri**, "Integrated simulation and formal verification of a simple autonomous vehicle", *International Conference on Software Engineering and Formal Methods*, pages:300–314, 2017. **Candidate's contributions**: Modelled the plant in Simulink, created the ad-hoc communication pattern, wrote the executable theory of the control, run co-simulation experiments.

5. C. Bernardeschi, M.Di Natale, G. Dini, **M. Palmieri**, "Verifying Data Secure Flow in AUTOSAR Models by Static Analysis", *ICISSP*, pages:704–713, 2017.

Candidate's contributions: Modelled the case study, investigated the formal notation.

6. C. Bernardeschi, A. Domenici, **M. Palmieri**, "Towards Stochastic FMI Co - Simulations: Implementation of an FMU for a Stochastic Activity Networks Simulator", *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages:34–44, 2018. **Candidate's contributions**: Created the FMI interface for Mobius, implemented the watertank model on Mobius, run co-simulation experiments.

7. C. Bernardeschi, A. Fagiolini, **M. Palmieri**, G. Scrima, F. Sofia, "ROS/Gazebo Based Simulation of Co-operative UAVs", *International Conference on Modelling and Simulation for Autonomous Systems*, pages:321–334, 2018. **Candidate's contributions**: Conceptualized the communication pattern, supervised the implementation of the communication pattern

8. **M. Palmieri**, H. D. Macedo, "Automatic Generation of Functional Mock-up Units from Formal Specifications", *Co-sim CPS 2019*, 2019. **Candidate's contributions**: (Conceptualization of the methodology proposed, implementation of the templates for FMU generation, informal verification of the proposed methodology)

9. C. T. Hansen, **M. Palmieri**, C. Gomes, K. Lausdahl, H. D. Macedo, N. Battle, P. G. Larsen, "Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks ", *Co-sim CPS 2019*, 2019. **Candidate's contributions**: Implementation of initial features of the new master algorithm, discussed the features of the proposed architecture

10. C. Bernardeschi, A. Domenici, **M. Palmieri**, "Modeling and Simulation of Attacks on Cyber-physical Systems", *ICISSP*, 2019. **Candidate's contributions**: Conceptualization of the formalization of the attacks, modeling the attacks in the case study, creation of the case study, analysis of the case study.

Journal papers

1. **M. Palmieri**, C. Bernardeschi, P. Masci , "A Framework for FMI-based Co-Simulation of Human-Machine Interfaces", *International Journal on Software and Systems Modeling*, 2019. **Candidate's contributions**: Designed, implemented and extended the framework, formally verified the framework, used the framework to create the case study, extended the case study with a new model, run co-simulations experiments and analyzed the outputs.

2. C. Vallati, S. Brienza, **M. Palmieri**, G. Anastasi, “Improving network formation in IEEE 802.15. 4e DSME”, *Computer Communications*, pages:01–09, 2017. **Candidate’s contributions**: Implemented the protocol, designed the experiments, investigated the proposed improvements, run simulations, performed statistical analysis on data collected by experiments.

3. C. Bernardeschi, M. Di Natale, G. Dini, **M. Palmieri**, “Verifying data secure flow in AUTOSAR models”, *Journal of Computer Virology and Hacking Techniques*, pages:269–289, 2018. **Candidate’s contributions**: Modeled the case study, investigated the formal notation, extended the formal notation, supervised the implementation of the tool.

Papers under review

1. C. Bernardeschi, A. Domenici, **M. Palmieri**, “Formalization and Co-simulation of Attacks on Cyber-physical Systems”, *Journal of Computer Virology and Hacking Techniques*. **Candidate’s contributions**: Conceptualization of the formalization of attacks, extended the formalization to include interactive attacks, modeling the attacks in the case study, analysis of the case study, formal verification of the case study, formal specification of the plant used in the case study.

Bibliography

Abel, A., Blochwitz, T., Eichberger, A., Hamann, P., and Rein, U. (2012). Functional mock-up interface in mechatronic gearshift simulation for commercial vehicles. In *Proceedings of the 9th International MODELICA Conference*, pages 775–780. Linköping University Electronic Press.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Arney, D., Goldman, J. M., Bhargav-Spantzel, A., Basu, A., Taborn, M., Pappas, G., and Robkin, M. (2012). Simulation of medical device network performance and requirements for an integrated clinical environment. *Biomedical Instrumentation & Technology*, 46(4):308–315.

Asarin, E., Dang, T., and Maler, O. (2002). The d/dt tool for verification of hybrid systems. In Brinksma, E. and Larsen, K. G., editors, *Computer Aided Verification*, pages 365–370, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

Behrmann, G., David, A., and Larsen, K. G. (2006). A Tutorial on UPPAAL 4.0. `http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf`.

Bequette, B. W. (2003). *Process control: modeling, design, and simulation*. Prentice Hall Professional.

Bernardeschi, C. and Domenici, A. (2016). Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System. *Inf. Process. Lett.*, 116(6):409–415.

Bernardeschi, C., Domenici, A., and Masci, P. (2018). A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. *IEEE Trans. Software Eng.*, 44(6):512–533.

Bernardeschi, C., Masci, P., Caramella, D., and Dell'Osso, R. (2019). The benefits of using interactive device simulations as training material for clinicians: an experience report with a contrast media injector used in CT. *SIGBED Rev., Special Issue on Medical Cyber-Physical Systems Workshop 2018 (MCPS'18)*, 16(2):41–45.

Blochwitz, T., Otter, M., Åkesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauß, J., Neumerkel, D., Olsson, H., and Viel, A. (2012). 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.

Bolton, M. L., Siminiceanu, R. I., and Bass, E. J. (2011). A systematic approach to model checking human–automation interaction using task analytic models. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 41(5):961–976.

Broenink, J. F. (1999). 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5-6):481–492.

Campos, J. C. and Harrison, M. D. (2009). Interaction engineering using the IVY tool. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 35–44. ACM.

Chaudemar, J.-C., Savicks, V., Butler, M., and Colley, J. (2014). Co-simulation of Event-B and Ptolemy II Models via FMI. In *ERTS 2014 "Embedded real time software and systems"*, Toulouse, FR.

Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. (2015). Hycomp: An smt-based model checker for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 52–67. Springer.

Clarke, E. M. (1997). Model checking. In Ramesh, S. and Sivakumar, G., editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg. Springer Berlin Heidelberg.

CNNNews (2018a). Tesla in autopilot mode crashes into fire truck. `http://money.cnn.com/2018/01/23/technology/tesla-fire-truck-crash/index.html`.

CNNNews (2018b). Uber self-driving car kills pedestrian in first fatal autonomous crash. `http://money.cnn.com/2018/03/19/technology/uber-autonomous-car-fatal-crash/index.html`.

Couto, L. D., Basagiannis, S., Ridouane, E. H., Mady, A. E.-D., Hasanagic, M., and Larsen, P. G. (2018). Injecting Formal Verification in FMI-Based Co-simulations of

Cyber-Physical Systems. In Cerone, A. and Roveri, M., editors, *Software Engineering and Formal Methods*, pages 284–299, Cham. Springer International Publishing.

De Moura, L. and Bjørner, N. (2011). Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77.

Dutertre, B. (1996). Elements of mathematical analysis in pvs. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96, pages 141–156, Berlin, Heidelberg. Springer-Verlag.

Franceschini, G. and Macchietto, S. (2008). Model-based design of experiments for parameter precision: State of the art. *Chemical Engineering Science*, 63(19):4846 – 4872. Model-Based Experimental Analysis.

Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O. (2011). Spaceex: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan, S. Q., editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer.

Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., and Broman, D. (2005). The Open Modelica Modeling, Simulation, and Development Environment. In *In Proceedings of the 46th Conference on Simulation and Modeling of the Scandinavian Simulation Society (SIMS2005)*, pages 83–90.

Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., and Platzer, A. (2015). Keymaer-aÂăx: An axiomatic tactical theorem prover for hybrid systems. In Felty, A. P. and Middeldorp, A., editors, *Automated Deduction - CADE-25*, pages 527–538, Cham. Springer International Publishing.

Garro, A. and Falcone, A. (2015). On the Integration of HLA and FMI for Supporting Interoperability and Reusability in Distributed Simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, DEVS'15*, pages 9–16. Society for Computer Simulation International.

Goderis, A., Brooks, C., Altintas, I., Lee, E. A., and Goble, C. (2007). Composing different models of computation in Kepler and Ptolemy II. In *International Conference on Computational Science*, pages 182–190. Springer.

Goldman, J. M. (2008). Medical devices and medical systems-Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE)-Part 1: General requirements and conceptual model. *ASTM International*.

Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. (2018). Co-simulation: A survey. *ACM Comput. Surv.*, 51(3):49:1–49:33.

Harrison, M. D., Freitas, L., Drinnan, M., Campos, J. C., Masci, P., di Maria, C., and Whitaker, M. (2019). Formal techniques in the safety analysis of software components of a new dialysis machine. *Science of Computer Programming*, 175:17–34.

Harrison, M. D., Masci, P., Campos, J. C., and Curzon, P. (2017). Verification of User Interface Software: the Example of Use-Related Safety Requirements and Programmable Medical Devices. *IEEE Transactions on Human-Machine Systems*, to appear.

Heitmeyer, C., Kirby, J., Labaw, B., and Bharadwaj, R. (1998). SCR: A toolset for specifying and analyzing software requirements. In *International Conference on Computer Aided Verification*, pages 526–531. Springer.

Henzinger, T. A. (1996). The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–292, Washington, DC, USA. IEEE Computer Society.

Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1997). Hytech: A model checker for hybrid systems. In Grumberg, O., editor, *Computer Aided Verification*, pages 460–463, Berlin, Heidelberg. Springer Berlin Heidelberg.

Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244.

Jensen, J. C., Chang, D. H., and Lee, E. A. (2011). A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1666–1671.

Krammer, M., Marko, N., and Benedikt, M. (2016). Interfacing Real-Time Systems for Advanced Co-Simulation-The ACOSAR Approach. In *Software Technologies: Applications and Foundations (STAF) Doctoral Symposium/Showcase*, pages 32–39.

Larsen, P. G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., and Verhoef, M. (2010). The overture initiative integrating tools for vdm. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6.

Larsen, P. G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., et al. (2016). Integrated tool chain for model-based design of Cy-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.

Leveson, N. (2011). *Engineering a safer world*. MIT Press.

Masci, P., Mallozzi, P., DeAngelis, F., Serugendo, G., and Curzon, P. (2015a). 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*.

Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P., and Thimbleby, H. (2015b). *PVSio-web 2.0: Joining PVS to HCI*, pages 470–478. Springer International Publishing.

Masci, P., Rukšenas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., and Thimbleby, H. (2015c). 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.

Masci, P., Zhang, Y., Jones, P., Curzon, P., and Thimbleby, H. (2014a). Formal verification of medical device user interfaces using PVS. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg.

Masci, P., Zhang, Y., Jones, P. L., Oladimeji, P., D'Urso, E., Bernardeschi, C., Curzon, P., and Thimbleby, H. (2014b). Combining PVSio with Stateflow. In *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, pages 209–214.

Mauro, G., Thimbleby, H., Domenici, A., and Bernardeschi, C. (2017). Extending a user interface prototyping tool with automatic MISRA C code generation. In Dubois, C., Masci, P., and Méry, D., editors, *Proceedings of the Third Workshop on Formal Integrated Development Environment, Limassol, Cyprus, November 8, 2016*, volume 240 of *Electronic Proceedings in Theoretical Computer Science*, pages 53–66. Open Publishing Association.

Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M., and Chamberlain, J. (2015). Daidalus: Detect and avoid alerting logic for unmanned systems. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 5A1–1–5A1–12.

Muñoz, C. (2003). Rapid prototyping in PVS. Technical Report NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA.

Nagele, T. and Hooman, J. (2017). Co-simulation of cyber-physical systems using HLA. In *Proceedings of the IEEE Computing and Communication Workshop and Conference*, CCWC'17, pages 1–6.

Oladimeji, P., Masci, P., Curzon, P., and Thimbleby, H. (2013). PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In *FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013*.

Olfati-Saber, R., Fax, J. A., and Murray, R. M. (2007). Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233.

Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *Automated Deduction — CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg.

Owre, S., Rushby, J., Shankar, N., and Von Henke, F. (1995). Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. (1999). PVS language reference. *Computer Science Lab., SRI International, Menlo Park, CA*, 1(2):21.

Pajic, M., Mangharam, R., Sokolsky, O., Arney, D., Goldman, J., and Lee, I. (2014). Model-driven safety analysis of closed-loop medical systems. *IEEE Transactions on Industrial Informatics*, 10(1):3–16.

Palensky, P., Meer, A. A. V. D., Lopez, C. D., Joseph, A., and Pan, K. (2017a). Cosimulation of intelligent power systems: Fundamentals, software architecture, numerics, and coupling. *IEEE Industrial Electronics Magazine*, 11(1):34–50.

Palensky, P., van der Meer, A., Lopez, C., Joseph, A., and Pan, K. (2017b). Applied cosimulation of intelligent power systems: Implementing hybrid simulators for complex power systems. *IEEE Industrial Electronics Magazine*, 11(2):6–21.

Paterno, F., Santoro, C., and Spano, L. D. (2009). MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4).

Pedersen, N., Bojsen, T., and Madsen, J. (2017). Co-simulation of Cyber Physical Systems with HMI for Human in the Loop Investigations. In *Proceedings of the Symposium on Theory of Modeling & Simulation*, TMS/DEVS '17, pages 1:1–1:12, San Diego, CA, USA. Society for Computer Simulation International.

Sander, I. and Jantsch, A. (2004). 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.

Son, H. S. and Seong, P. H. (2003). Development of a safety critical software requirements verification method with combined CPN and PVS: a nuclear power plant protection system application. *Reliability Engineering & System Safety*, 80(1):19 – 32.

Thimbleby, H. (2010). *Press on: principles of interaction programming*. The MIT Press.

Thule, C., Lausdahl, K., Gomes, C., Meisl, G., and Larsen, P. G. (2019). Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45 – 61.

Wang, B. and Baras, J. S. (2013). 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.

Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer*, 23(9):8–22.

Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., De Angelis, F. L., Serugendo, G. D. M., Dobson, S., Fernandez-Marquez, J. L., Ferscha, A., Mamei, M., et al. (2015). Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 17:236–252.

Zeyda, F., Ouy, J., Foster, S., and Cavalcanti, A. (2018). Formalising cosimulation models. In Cerone, A. and Roveri, M., editors, *Software Engineering and Formal Methods*, pages 453–468, Cham. Springer International Publishing.