Stochastic Hybrid Systems Meet Software Components for Well-Founded Cyber-Physical Systems Software Architectures

Jacques Malenfant
Jacques.Malenfant@lip6.fr
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

ABSTRACT

Cyber-physical control systems (CPCS) are notoriously difficult to specify, implement, test, validate and verify. In this paper, we propose to integrate hybrid systems, and their declensions as hybrid automata and DEVS simulation models, within a full-fledged and well-founded software component model tailored for CPCS. The key concept is to attach to components modular, composable and reusable behavioural and simulation models. The goal is to seamlessly support the software development process, from model-in-the-loop initial validation, until deployment time actual system verification. The resulting comprehensive modeling and software implementation tool aims at fully supporting the different phases of the software life cycle to provide more reliable, robust, reusable and adaptable CPCS using less resources.

KEYWORDS

hybrid systems, software components, simulation, test, validation

ACM Reference Format:
Jacques Malenfant. 2019. Stochastic Hybrid Systems Meet Software Components for Well-Founded Cyber-Physical Systems Software Architectures. In *European Conference on Software Architecture (ECSA), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3344948.3344989

1 INTRODUCTION

Cyber-physical control systems (CPCS) are well-known to be very difficult to fully specify, implement, test, verify and validate. CPCS bring computational systems and physical world phenomena tightly together, which they sense, control and actuate. CPCS have grown in complexity to a point where developing correct software architectures now requires novel software engineering techniques and processes. Hence, we are in urgent need for specific robust software models and engineering processes.

Appropriately modeling CPCS requires behavioral models that capture both the discrete nature of computational systems and the continuous one of the physical world. Hybrid systems have been developed in the last decades to provide such a capability. However, this research has not yet reached the realm of software engineering for CPCS. To fill this gap in a component-based approach, hybrid systems need not only to capture the overall application behaviour but also to be constructed from the composition of models attached to each component to follow the compositional nature of the latter.

In this paper, we propose to integrate a form of modular hybrid systems and software component models into a well-founded software model for CPCS. The ultimate goal is to support a software development process dedicated to CPCS by addressing issues such as: (1) comprehensive behavioral specification from modular and reusable models composed in parallel with their components, (2) model-in-the-loop simulation for initial system validation, (3) unit and integration testing as well as (4) software verification and validation through software-in-the-loop simulation, (5) deployment time system identification and control law synthesis, (6) hardware-in-the-loop simulation for system validation and verification and (7) run time CPCS self-adaptation. Currently, items (1) and (2) are tackled and illustrated in this paper, the remaining ones still being future work.

In the rest of the paper, we first look at stochastic hybrid systems as proposed by mathematics. Then we introduce automata models of hybrid systems providing a modular modeling scheme. We next introduce modular simulation modeling and our component model for CPCS. The paper ends with a survey of the related work followed by a conclusion and a discussion of our perspectives. Throughout the paper, a real-world example is used to illustrate our concepts up to actual results of model-in-the-loop simulations.

2 HYBRID SYSTEMS

We now present hybrid systems [4, 5, 10] as modeling tool for CPCS.

2.1 Mathematical models of hybrid systems

Mathematicians have proposed several general models of controlled hybrid systems. Branicky's one [2] partitions the hybrid state space $\mathcal{S} = \bigcup_{q \in Q} X_q \times \{q\}$ into a countable set of discrete states $Q = \{q_0, q_1, \ldots\}$, each of them defining a continuous state space $X_q, q \in Q$. Jumps between discrete states occur upon events, either changes in the value of discrete variables or frontier conditions met by continuous variables. The continuous dynamics, controlled by control laws U_q within each discrete state q, is defined by a set of equations f_q that may be of different types, but often algebraic equations or differential ones such as:

$$\dot{\mathbf{x}}(t) = f_q(\mathbf{x}(t), u_q(t)), \mathbf{x} \in X_q$$

As many real systems exhibit random behaviours, stochastic extensions to hybrid systems have been proposed [11]. Randomness can show up in different ways, such as: (1) stochastic continuous behaviours modeled as brownian motions and stochastic differential equations [15] or (2) stochastic jump transitions where the hybrid

ECSA, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7142-1/19/09…$15.00
https://doi.org/10.1145/3344948.3344989

states after transitions follow density probability functions that depend upon the current hybrid state and the control signal.

2.2 Case study

Our illustrative case study is an application where a portable computer exchanges large amounts of data with a server through a WiFi connection [14]. When the WiFi bandwidth is low, the system adapts itself by compressing the data to use less bandwidth. However, compression uses more PC processor, which consumes more battery. Hence, when the bandwidth is high or the battery is too low, it stops compressing. The objective is to get the best possible data transfer rate, including the compression and decompression times, while maintaining the PC alive as long as possible. For the sake of simplicity, we only model individual runs to battery exhaustion and we consider only the components that are exchanging the data. Note that our goal is to illustrate the use of hybrid systems modeling for CPCS, not to provide a faithful and complete model for this application.

Two variables are under the influence of the control:

- d: the data transfer rate in kbits/s.
- b : the battery level in mAh.

Around these variables, the model also defines:

- p is the bandwidth of the WiFi network in kbits/s.
- r_c is the rate of compression in kbits/s.
- r_u is the rate of uncompression in kbits/s.
- τ_c is the mean compression factor, $0 < \tau_c < 1$.

The dynamics of the system exhibits two important behaviours. First, the evolution of the remaining level of energy in the PC battery, which has three modes depending on whether compression is used, with draining rate ΔB_c in mAh/s, or not with draining rate ΔB_{nc} or the network is not accessible, with a draining rate ΔB_0 (all three assumed to be deterministic):

$$\dot{b}(t) = -\Delta B_c \qquad \dot{b}(t) = -\Delta B_{nc} \qquad \dot{b}(t) = -\Delta B_0$$

In the runs presented later, we chose $\Delta B_c = 1.5\, mAh/s$, $\Delta B_{nc} = 1.0\, mAh/s$ and $\Delta B_0 = 0.5\, mAh/s$. Second, the bandwidth is assumed to be stochastic and, for the sake of simplicity, to follow a brownian motion expressed by a stochastic differential equation of the form:

$$\dot{p}(t) = \sigma(p(t))dB(t)$$

We know intuitively that a threshold bandwidth exists over which the data transfer rate is higher without compression and under which it is higher with compression. From the data transfer rates equations in the two cases, a few algebraic manipulations show that this threshold p_s is:

$$p_s = \frac{(1-\tau_c)r_c r_u}{r_d + \tau_c r_u}$$

For example, for $r_c = 50$ kbits/s, $\tau_c = 0.4$, $r_u = 75$ kbits/s, $p_s \approx 23.68$ kbits/s. A simple control law with hysteresis can be adopted for the decision on the compression mode:

- If $p \geq P_{sup} > p_s$, go to the non compression mode;
- If $p \leq P_{inf} < p_s$, go to the compression mode.

For this example, we may use $P_{sup} = 25$ kbits/s and $P_{inf} = 21$ kbits/s. To get a longer autonomy on battery, we use a simple control law with a threshold B:

- If $b \geq B$, authorise the compression.
- If $b < B$, forbid the compression.

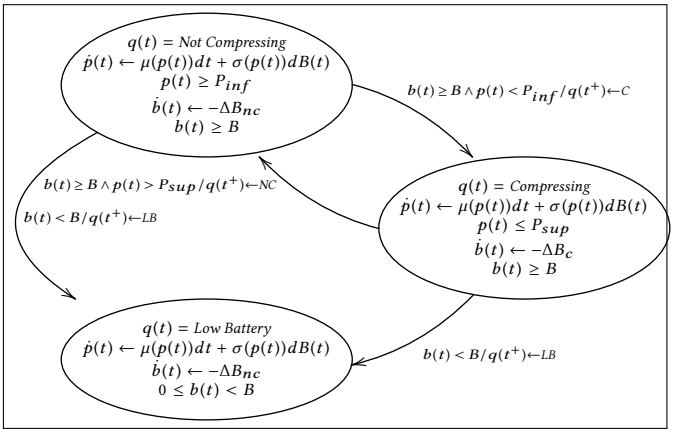

Figure 1: Hybrid system baseline model for the data transfer use case without WiFi network interruptions.

For example, if the battery has a capacity of $B_{max} = 7500$ mAh, the threshold can be set to $B = B_{max}/2 = 3750$ mAh.

The Figure 1 presents a stochastic hybrid systems model for this use case where no network interruption can occur. The notation uses an automaton representation exhibiting several discrete modes of different continuous behaviours with differential equations and discrete transitions triggered by conditions on the variables. Discrete states are identified by the values of the discrete variable q, which can take three values: *Compressing*, *Not Compressing* and *Low Battery*. When a transition occurs, variables can be set to "initial" values in the new hybrid state. Hence, a transition has a trigger part and a variable assignment part, separated by "/". As states can also express assertions on their variables, the symbol "←" means assignment while "=" means equality in assertions. When resetting variables at time t, the notations t^- and t^+ mean the value right before and right after the discrete transition, respectively.

3 HYBRID AUTOMATA

Figure 1 presents a monolithic model, mixing control and baseline behaviours and which rapidly becomes heavy for complex systems. Hybrid automata allow to break complex models into smaller composable ones, hence providing modularity and reusablility.

3.1 HIOA and TIOA

Henzinger [6, 7] as well as Lynch and her team [9, 12] were pioneers in the definition and study of *hybrid automata*. Lynch team's work provides a little more of the modularisation features we are looking for. First they define a hybrid automaton as follows [12]:

Definition. A hybrid automaton (HA) $\mathcal{H}$ is a tuple $(W, X, Q, \Theta, E, H, D, \mathcal{T})$ such that:

- A set W of external variables and a set X of internal variables, disjoint from each other. Define $V \triangleq W \cup X$.
- A set $Q \subseteq val(X)$ of states.
- A nonempty set $\Theta \subseteq Q$ of start states.
- A set E of external actions and a set H of internal actions, disjoint from each other. Define $A \triangleq E \cup H$.
- A set $D \subseteq Q \times A \times Q$ of discrete transitions.

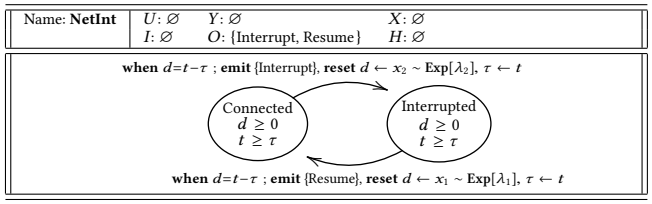

Figure 2: Network interruption model presented in an easy to read HIOA/TIOA format (*U*: imported continuous variables, *Y*: exported continuous variables, *X*: internal continuous variables; *I*: imported actions, *O*: exported actions, *H*: internal actions).

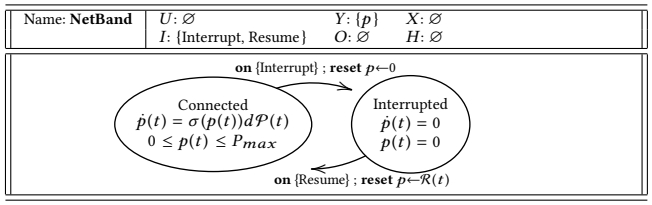

Figure 3: Network bandwidth model with interruptions.

- A set $\mathcal{T}$ of trajectories $\tau(t)$ for V such that the values of the variables in X remain in Q for all $t \in dom(\tau)$. □

Lynch *et al.* [12] distinguish continuous variables from actions, assumed discrete. Other models use the term event for action. For modeling purposes, we adopt a model of HA where actions are expressed either as modifications of discrete variables triggering transitions or by emitting and receiving events in transitions with expressions **emit** and **on**, respectively. Lynch *et al.* define the composition $\mathcal{H}_1 \parallel \mathcal{H}_2$, (and for this the compatibility of $\mathcal{H}_1$ and $\mathcal{H}_2$) by merging their variables, transitions and trajectories, if they respect rules such as not sharing internal variables.

With the above definition, composing two HA imposes that their external variables defined in both of them with the same name represent in fact the same variable, which implies that the two HA define the same trajectory for them. To avoid this complexity, Lynch *et al.* introduce hybrid I/O automata, which further distinguish among their external variables and actions between imported and exported ones, and then impose a unique producer for each of them.

Definition. A hybrid I/O automaton (HIOA) $\mathcal{A}$ is a tuple $(\mathcal{H}, U, Y, I, O)$ where:

- $\mathcal{H} = (W, X, Q, \Theta, E, H, D, \mathcal{T})$ is a hybrid automaton.
- U and Y partition W into input and output variables, resp.
- I and O partition E into input and output actions, resp. □

The next step is to define the composition of two HIOA:

Definition. Let $\mathcal{A}_1 = (\mathcal{H}_1, U_1, Y_1, I_1, O_1)$ and $\mathcal{A}_2 = (\mathcal{H}_2, U_2, Y_2, I_2, O_2)$, be two hybrid I/O automata, they are compatible if $\mathcal{H}_1$ and $\mathcal{H}_2$ are compatible and if $Y_1 \cap Y_2 = O_1 \cap O_2 = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible, their composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ is the tuple $\mathcal{A} = (\mathcal{H}, U, Y, I, O)$ where $\mathcal{H} = \mathcal{H}_1 \parallel \mathcal{H}_2$ and:

- $Y = Y_1 \cup Y_2$,
- $O = O_1 \cup O_2$, and
- $U = (U_1 \cup U_2) \cap Y$,
- $I = (I_1 \cup I_2) \cap O$. □

HIOA do not solve all the modularity concerns to model software architectures. Indeed, sharing continuous variables does not account for digital network communication, where only discretised values can be punctually exchanged. To cater for this restriction, Lynch's team introduces timed I/O automata [9] *i.e.*, HIOA with no external continuous variables:

Definition. A timed I/O automaton (TIOA) is a hybrid I/O automaton $\mathcal{A} = (\mathcal{H}, U, Y, I, O)$ where $U = \emptyset$ and $Y = \emptyset$. □

While HIOA express tightly coupled models of *centralised* systems, TIOA express models of *decentralised/distributed* ones.

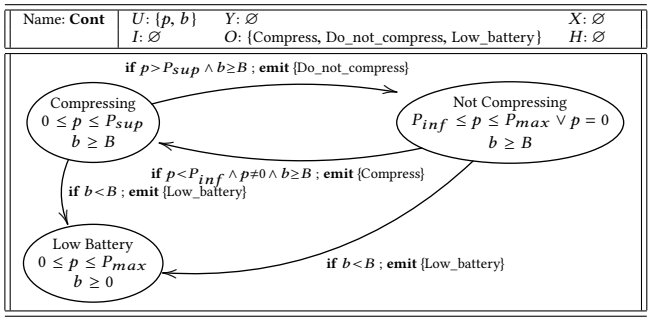

Figure 4: Controller models (portable computer and server).

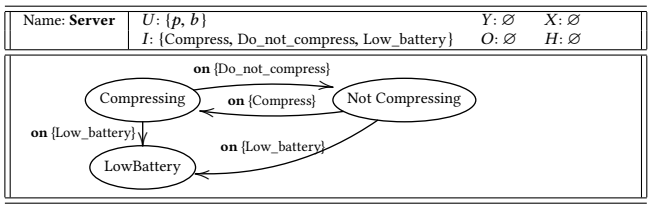

Figure 5: Server model.

3.2 Case study: continued

With HIOA, we can decompose our model to reveal actual components. Begin with the network bandwidth. It is composed of a continuous model and an interruption model, similar to a failure model. The Figure 2 presents a simple network interruption model where the time between interruptions follows an exponential distribution with mean λ_1 and where the duration of interruptions also follows an exponential distribution with mean λ_2. This model emits events Interrupt and Resume that are imported by a network bandwidth model with interruptions shown in the Figure 3. In this model, the continuous bandwidth follows a brownian motion expressed as a stochastic differential equation and the bandwidth at resumption follows a Beta distribution:

$$d\mathcal{P}(t) \sim \mathbf{Exp}[\lambda_p] \in [0, \infty[$$

$$\lambda_p = \text{average first derivative of the bandwidth}$$

$$\sigma(p) = \begin{cases} -1 & \textbf{if } u < p/P_{max} \\ 1 & \textbf{if } u \geq p/P_{max} \end{cases}$$

$$\text{with } u \sim \mathbf{U}[0,1] \in [0,1]$$

$$\mathcal{R}(t) = rP_{max}, \text{with } r \sim \mathbf{Beta}[\alpha_p, \beta_p] \in [0,1]$$

This network bandwidth model exports the variable p *i.e.*, the WiFi bandwidth.

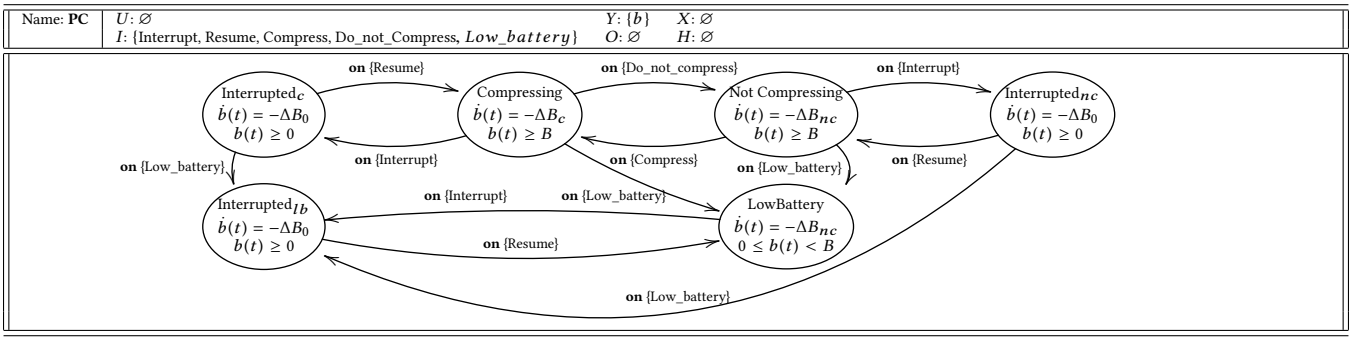

Figure 6: Portable computer model.

The Figure 4 presents the controller model that imports the variables p and b and given their values makes the decisions about the compression and no compression. It does so by exporting the events Compress, Do_not_compress and Low_battery encoding the decisions. This model is used both on the server and the PC sides in order to produce a complete model that can be distributed.

The server model of the Figure 5 imports these events and use them to keep track of the current mode (in an implementation, this would result in changing the data emission and reception to add or retract the compression modules). The Figure 6 presents the portable computer model that, as the server one, imports the events from the controller to follow the decided mode but also produces the variable b as it models the artefact that holds the actual battery.

From centralised to decentralised models

A complete model is obtained by composing the six HIOA models with appropriate connections from exported events and variables to imported ones. But this composite model is still centralised for two reasons. First, the network bandwidth and the portable computer models export the variables p and b respectively which are imported by the controller models. If the controllers have to be implemented by distinct components, they would have to share continuous variables with the two other components, something that cannot be achieved in reality. Also, if the server controller is distant from the portable computer, the value of b cannot be shared instantaneously but rather have to be sent through a digital network with some (stochastic) delay.

To model these and turn HIOA models into TIOA, we introduce sensor and network transmission models (not shown here). Sensor models are composed of a Tic model that emits tic events at a regular pace, which are consumed by a sensor model that imports a continuous variable and emits events with a punctual value of this variable at each tic event reception, thus effectively discretising the continuous variable into a flow of events. Such sensor models are added to the network bandwidth and to the portable computer HIOA models to obtain TIOA models exchanging events only. The controller models are modified to take into account the sharing of events rather than continuous variables (not shown here due to space limitations).

Also, a network transmission model is introduced that imports and exports the same events but with a delay that follows some probability distribution (in our runs, a distribution $\mathbf{Gamma}(\kappa, \theta)$ with $\kappa = 11$, $\theta = 2$ and a mean of 22 msec).

4 CYBER-PHYSICAL SOFTWARE COMPONENT MODEL

Stochastic hybrid systems are very effective to model real systems, but they are very complex to use formally (proofs, model-checking, ...). However, they can be used to simulate the system in order to debug, test and systematically verify CPCS. As a proof-of-concept, we have developed a DEVS simulation library where models can be created, composed and included into components of a distributed component model in Java called BCM4Java [13].

4.1 From hybrid systems to simulation models

From a semantics point of view, stochastic hybrid systems express behaviours in a *declarative* way, aiming at formally proving properties: as models, they say *what* is the behaviour but not *how* to use it. Another use of stochastic hybrid systems is to simulate them *i.e.*, to compute trajectories of states and variables as samples of realistic runs of the system. Hence, as simulation models, stochastic hybrid systems express an *operational* semantics *i.e.*, *how* to perform these computations by employing simulation engines and defining simulation models that these engines can execute directly.

In simulation, DEVS [17] offers modular modeling capabilities. DEVS is based on discrete event simulation, but can be applied to continous simulations discretised through integration steps. DEVS splits simulation models into (1) *atomic models* representing baseline models that execute transitions between states and input and output events and (2) *coupled models* composing models by connecting outputs of ones to inputs of others and then coordinating their transitions. Each model can have its own simulation algorithm, but DEVS defines a unique *simulation protocol* through which models are coordinated and activated repeatedly to perform their next transition until the end of the run while ensuring the right order for the transitions and the communication of the external events. DEVS admits different ways to coordinate models:

- for MIL, an explicitly synchronised protocol where coordinators enforce a global simulation clock by distributing events and activating model transitions in strict order;
- for SIL and HIL, an implicitly synchronised one forcing the model local clocks to strictly follow the real time [16] while models exchange events directly as messages.

We have implemented a DEVS simulation library in Java tailored to our needs in component-based software architectures. Each

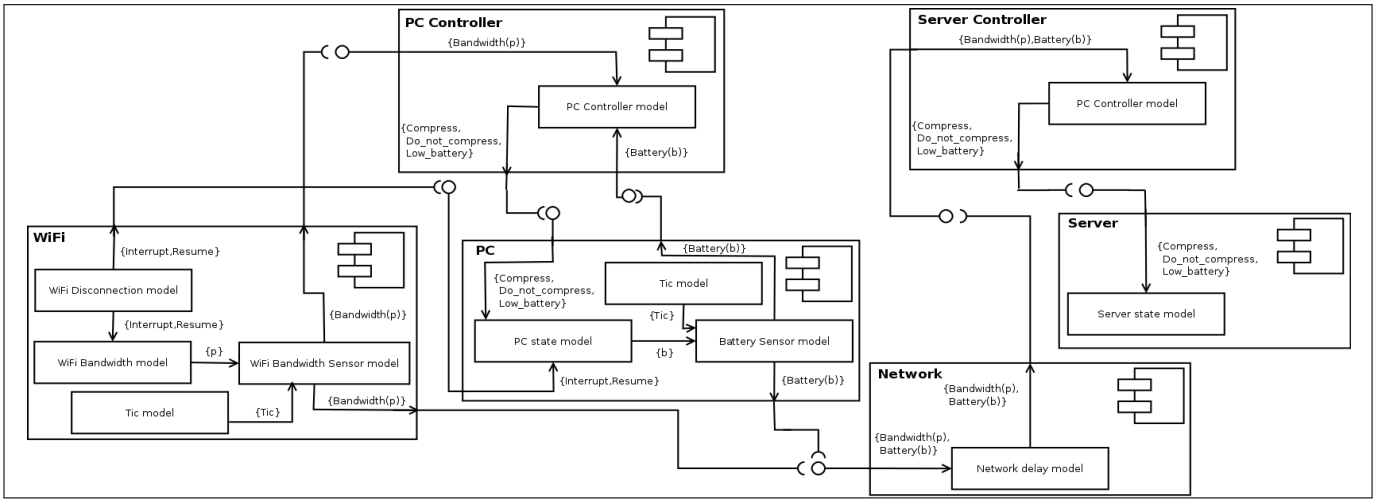

Figure 7: Component-based software architecture for the case study at the development stage where a model-in-the-loop simulation composite model is deployed.

model is created as a class from its definitional HIOA/TIOA with specific DEVS-based structures and behaviours so that they can be algorithmically composed from an architectural description. This provides both flexibility and reusability of simulation models. Simulation architectures can then be run by passing them run parameters and, at the end, they return simulation reports including actual realisations of the variables and statistics. The integration with the component model is implemented as reusable plug-ins, hence the programmer has no code to produce except a few ones for the instantiation, initialisation and launching the simulation runs.

4.2 From simulation models to CPS software components

Control engineering tells us that three major types of simulations are useful in CPCS development: (1) model-in-the-loop (MIL) simulations using models only to assess the correctness of the specification; (2) software-in-the-loop (SIL) simulations using models with the software to validate it through unit and integration testing; and (3) hardware-in-the-loop (HIL) simulations using models for the environment only hence targeting the actual system.

Our objective is to support these three kinds of simulations to cater for a *seamless* process going from a MIL simulation at the earlier stages to validate the specification of the system, to SIL simulations to perform unit testing, and then integration testing of the software during the development and to HIL simulations, both at design and deployment time, to verify the system. To achieve this goal, we build the simulation capability into the software architecture by making components able to hold and execute DEVS models packaged as plug-ins that can be easily switched on and off. They implement the DEVS models and simulation engines and automatise the deployment of the simulation architecture as well as the execution of runs, the gathering and analysis of the reports.

The conceptual approach is to attach a (perhaps composite) model to each component in the architecture and to connect the corresponding simulation models when connecting the components. In our component model [13], components can be distributed among different hosts, hence a connection between two components can use a digital network. Therefore, exchanges among their simulation models can only be events. Hence, at the component level, the simulation models must represent TIOA. However, these TIOA can be obtained by composing both HIOA and TIOA.

4.3 Case study: continued

The Figure 7 illustrates how HIOA/TIOA are turned into DEVS models individually implemented then composed and finally deployed over their corresponding software components. The Figure presents the architecture at the MIL simulation stage *i.e.*, early in the development process when simulations are used to verify and validate the specification. The four major components are implementing the PC side and server side data exchangers and their respective controllers. Inside each component, simulation models can represent HIOA and TIOA composed to obtain a TIOA. Imported and exported events can be exchanged with models of other components and for that purpose, usual component connections through ports are used. Arrows show the flow of events and variables among models and events sent from a component to another one.

For MIL as well as unit and integration testing through SIL simulations, the environment model capturing the WiFi bandwidth evolution over time is implemented as a simulation model in a WiFi component acting as a test stub. Also, the Network component is another test stub introduced only for the purpose of MIL and SIL simulations to ease the testing. Indeed, for HIL, the actual network would replace this stub.

In the current implementation, atomic models are programmed as subclasses of classes like `AtomicModel` and `AtomicHIOA` where the main methods to be defined (to follow the DEVS protocol) are for “internal step” that computes the new state of the model after the transition and for “time advance” that computes the time to wait until the next transition. Other methods are also implemented

to initialise the state at the beginning of the simulation as well as to cater for the simulation report and the plotting of the results. The composition of atomic models into coupled model needs simpler classes to represent them, mainly to declare the imported and exported events. In this use case, atomic models are defined by classes having around 100 lines of core code (actual model execution) and 200 lines of supporting code (simulation report creation and plotting). Then, simulation architecture descriptions, defining how models are composed, are used by the composition algorithm to create actual model instances and connect them. The entire use case architecture description is roughly 200 lines of code.

The Figure 8 shows plots produced by a run lasting 5000 seconds in simulated time, enough to see the controllers switching to the LowBattery mode after approximately 3750 seconds. This run exhibits several network interruptions as the mean time between interruptions and their mean duration were set to 200 sec. and 10 sec. respectively. This screen shot is not precise enough to clearly see the difference between the WiFi bandwidth model and its sensor one; the first is discretised by the integration step, which is much smaller than the sensing rate of the second (hence some very short interruptions go unnoticed to the sensor). A similar effect appears between the Battery level model and its sensor one. The two controller models impose the state transitions of their corresponding component (PC and server), which follow them. Only the PC state model shows its transition to the network interrupted mode, used as we have seen to model a slower battery draining because no network communication can occur in them.

5 RELATED WORK

Masaccio [7] proposes a component-oriented reformulation of hybrid automata but where "components" are modular models, not software ones. Co-simulation, understood as the joint execution of a simulation with a software system, has been proposed to provide a form of SIL [1, 16, 18] but seen as separate entities rather than merged into a unifying concept. Robotics has produced an extensive literature on the joint use of software/hardware and simulation to test robots too large to cite here. Hence, none of these works consider the integration of software components and modular simulation models, but rather keeps them separated.

Most of the related works on the simulation of CPCS target MIL for verification, yet some also consider SIL for software testing. However, none tries to define a full-fledged testing process. The most comprehensive work on this subject we know of is the Zohaib Iqbal's *et al.* [8], though they only consider discrete systems and keep software and simulators separated. De Roo *et al.* [3] propose the only work that we found addressing the integration of software and simulation, but they only use the continuous part of the modeling language and do not tackle the composability of models.

6 CONCLUSIONS AND PERSPECTIVES

We have discussed and illustrated how stochastic hybrid systems, hybrid automata and DEVS simulation models can be leveraged to propose a software component model tailored for CPCS and we illustrated it through a real-world case study. Such a component model can support a full-fledged software engineering methodology for CPCS addressing many crucial issues for practioners:

- formal specification with a strong behavioural modeling approach providing both compositionality and reusability;
- initial validation and verification through MIL simulations;
- software development, debugging, unit and integration testing using SIL simulations;
- control system identification, control law synthesis, configuration, validation and verification using HIL simulations;
- run-time self-adaptation to model changes.

We are currently developing this component model for distributed CPCS in Java with integrated HIOA/TIOA and DEVS modeling and simulation capabilities. The baseline component model BCM4Java is already available on GitHub [13]. The modeling and simulation extension is still under development (around 25.000 lines of code and documentation). When the CPCS component model will be fully completed, we plan first to attack the software engineering processes *per se*, developing proof-of-concepts for the different functionalities listed before. Our longer term goals are to research this approach for large-scale CPCS, like smart grids.

REFERENCES

[1] Ahmad T. Al-Hammouri, Michael S. Branicky, and Vincenzo Liberatore. 2008. Co-simulation Tools for Networked Control Systems. In *Proc. of Hybrid Systems: Computation and Control (Lecture Notes in Computer Science)*, Vol. 4981. Springer-Verlag, 16–29.

[2] Michael S. Branicky. 1995. *Studies in Hybrid Systems: Modeling, Analysis, and Control.* Ph.D. Dissertation. MIT.

[3] Arjan de Roo, Hasan Sözer, and Mehmet Aksit. 2014. Composing domain-specific physical models with general purpose software modules in embedded control software. *Software and Systems Modeling* 13 (2014), 55–81.

[4] Magnus Egerstedt. 2000. Behavior Based Robotics Using Hybrid Automata. In *Proc. of Hybrid Systems: Computation and Control (Lecture Notes in Computer Science)*, Vol. 1790. Springer-Verlag, 103–116.

[5] W.P.M.H. Heemels, D. Lehmann, J. Lunze, and B. De Schutter. 2009. *Introduction to hybrid systems*, 3–30. In Lunze and Lamnabhi-Lagarrigue [10].

[6] Thomas A. Henzinger. 1996. *The Theory of Hybrid Automata.* Technical Report. Electrical Engineering and Computer Sciences, University of California at Berkeley. updated version of a paper published by the same author at LICS 1996, pp. 278–292.

[7] Thomas A. Henzinger. 2000. Masaccio: A Formal Model for Embedded Components. In *IFIP TCS 2000 (Lecture Notes in Computer Science)*, Vol. 1872. Springer-Verlag, 549–563.

[8] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. 2015. Environment modeling and simulation for automated testing of soft real-time embedded software. *Software and Systems Modeling* 14 (2015), 483–524.

[9] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS'03)*. 166–177.

[10] Jan Lunze and Françoise Lamnabhi-Lagarrigue (Eds.). 2009. *Handbook of Hybrid Systems Control.* Cambridge University Press.

[11] J. Lygeros and M. Prandini. 2009. *Stochastic hybrid systems*, 249–276. In Lunze and Lamnabhi-Lagarrigue [10].

[12] Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Hybrid I/O Automata. *Information and Computation* 185 (2003), 105–157.

[13] J. Malenfant. 2018. BCM4Java. Available on GitHub at https://github.com/malenfantj/BCM4Java.git.

[14] Jacques Malenfant, Maria-Teresa Segarra, and Françoise André. 2001. Dynamic Adaptability: the MolèNE Experiment. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Reflection 2001 (Lecture Notes in Computer Science)*, A. Yonezawa (Ed.), Vol. 2192. Springer-Verlag, 110–117.

[15] Bernt Øksendal. 2013. *Stochastic Differential Equations* (6th ed.). Springer.

[16] Hessam S. Sarjoughian, Soroosh Gholami, and Thomas Jackson. 2013. Interacting Real-Time Simulation Models and Reactive Computational-Physical Systems. In *Proc. of the 2013 Winter Simulation Conference*. 1120–1131.

[17] Bernard P. Zeigler and Hessam S. Sarjoughian. 2013. *Guide to Modeling and Simulation of Systems of Systems.* Springer.

[18] Zhenkai Zhang, Emeka Eyisi, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. 2013. Co-Simulation Framework for the Design of Time-Triggered Cyber Physical Systems. In *Proc. of ACM ICCPS'13*. 119–128.

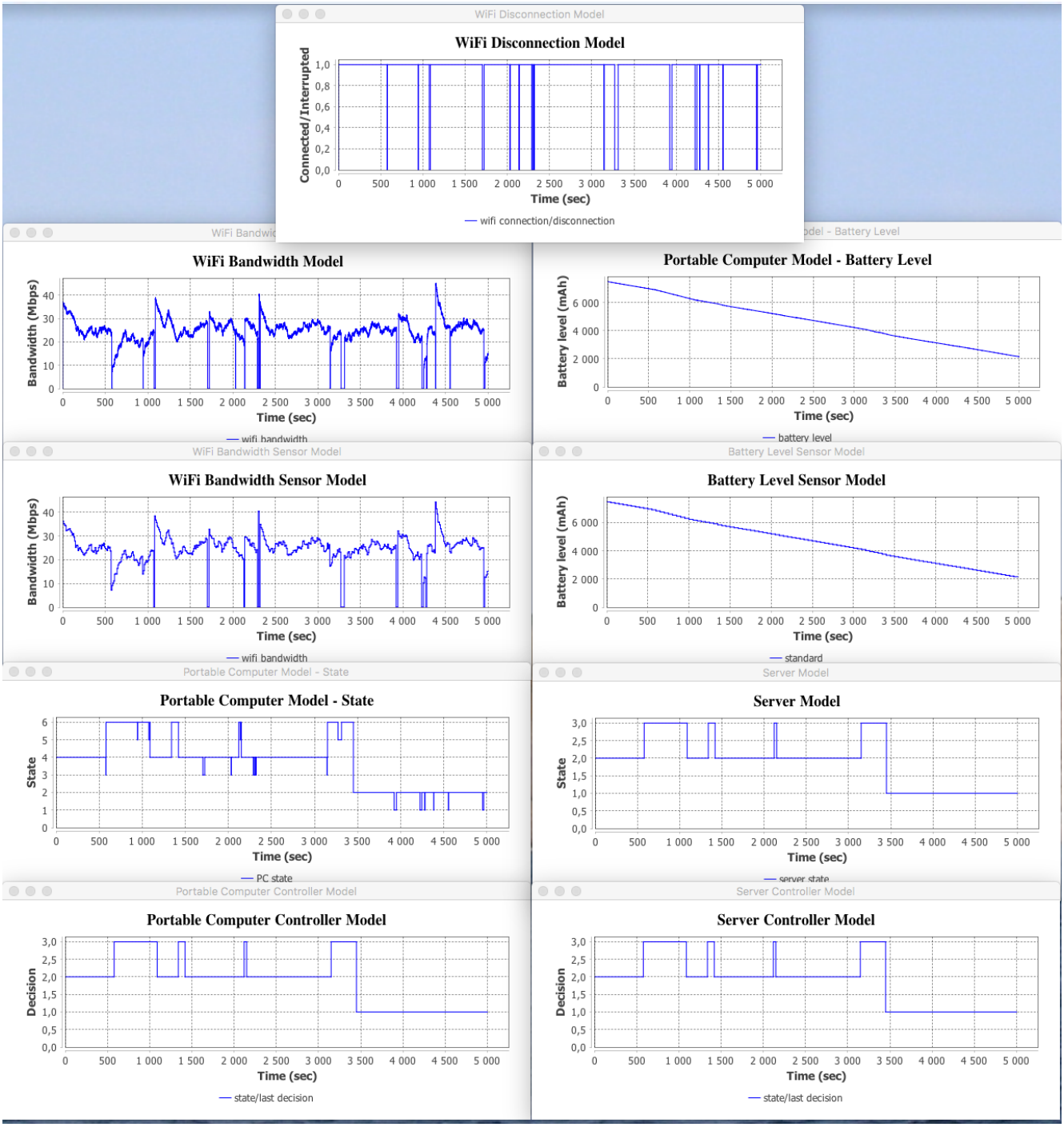

Figure 8: MIL simulation run results plotted for each model in our case study. The run lasts 5000 seconds (in simulated time), the elapsed simulation time appearing as the abcissa on all of the plots. At the top, the WiFi disconnection model generates interruption and resumption events, hence when the ordinates is 1.0, the WiFi is running but when 0.0 it is interrupted. On the left, the WiFi Bandwidth is shown in Mbps on both the bandwidth model and the bandwidth sensor one. The Portable computer model is shown in two plots: the battery level as a continuous variable obtained from computing its equations and the state (6 is compressing, 4 is non compressing and 2 is LowBattery, 1, 3 and 5 corresponding to network interruption states). The Battery sensor model is the discretisation of the battery level. The server model shows only one plot of its state that is similar and in fact follows the portable computer state (here 3 is compressing, 2 is non compressing and 1 is LowBattery). Finally the two controllers plot their decisions also as state transitions (the same, as they implement the same decision model).