Chapter 3
An Hourglass-Shaped Architecture for Model-Based Development of Networked Cyber-Physical Systems

Muhammad Umer Tariq and Marilyn Wolf

3.1 Introduction

Many technological achievements have been enabled by the field of *feedback control systems*, which deals with the process of controlling a physical system through a feedback controller. If the feedback controller is implemented as a *real-time computer system*, the resulting configuration of the feedback control system is referred to as *embedded control system*. Some prime examples of embedded control systems are automotive systems, avionics systems, and smart grid. The typical development process of an embedded control system can be partitioned into two distinct stages: controller design and controller implementation. During the controller design stage, a control systems engineer models the physical plant, derives the feedback control law, and validates the controller design through mathematical analysis and simulation. During the controller implementation stage, a computer systems engineer implements the feedback controller as a real-time computer system.

The field of *embedded control systems* brings together the fields of *control theory* and *real-time computer systems*. However, as noted in [15], the fields of *control theory* and *real-time computer systems* typically employ two completely different types of models: analytical models and computational models. As a result, two vastly different design processes are currently popular for the two stages of embedded control system development process: feedback controller design and feedback controller implementation as real-time computer system. Due

M. U. Tariq (✉)
ProsumerGrid, Inc., Atlanta, Georgia
e-mail: mumertariq@prosumergrid.com

M. Wolf
Georgia Institute of Technology, Atlanta, Georgia
e-mail: marilyn.wolf@ece.gatech.edu

M. A. Al Faruque, A. Canedo (eds.), *Design Automation of Cyber-Physical Systems*, https://doi.org/10.1007/978-3-030-13050-3_3

to the inherent differences between the abovementioned two stages, currently popular development methodologies for embedded control systems support very few correct-by-construction properties and depend heavily on testing the final implementation for creating confidence in the correct operation of an embedded control system under various runtime operating conditions. Therefore, current development techniques for embedded control systems are not capable of efficiently handling the ever-increasing complexity of these systems.

These limitations of the traditional embedded control system development techniques have created interest in taking a fresh look at the abstractions used in the traditional embedded control systems development process, resulting in a new field, *cyber-physical systems* (CPS) [39, 40]. The aim of CPS research is to develop an integrated theory as well as an integrated development toolset for controller design and controller implementation phases of the embedded control system development process. The hope is that this CPS research will enable the cost-effective development and maintenance of more complex versions of embedded control systems.

Recent CPS research efforts can be divided into two major categories: *platform-imperfection-aware feedback controller design* and *CPS-friendly computing platform design*. Under the category of *platform-imperfection-aware feedback controller design*, theoretical developments from the fields of hybrid systems [3], switched systems [21], time-delay systems [7], networked control systems [41], multi-agent networked systems [29], and game theory [16] are leveraged to develop a feedback controller design that takes into account the imperfections of the runtime computing platform (such as communication delays or failures caused by communication network congestion or cyber security attacks) at the design time [37]. The resulting "platform-imperfection-aware" feedback controller is either robust against the imperfections of runtime computing platform or possesses the capability to switch between different *control modes* to overcome the imperfections of runtime computing platform. Under the category of *CPS-friendly computing platform design*, CPS research has focused on specialized runtime computing platforms that have more predictable timing performance or provide correct-by-construction composition of software components. Some examples of this approach are provided in [17, 19, 22].

Model-based development (or model-driven development) of cyber-physical systems has the potential to bind the abovementioned CPS research efforts into an integrated, cross-layer CPS development methodology. In model-based development paradigm, high-level or platform-independent models (PIM) are transformed into lower-level or platform-specific models (PSM) through the process of model transformation. Both high-level and lower-level models are described using their own domain-specific modeling languages (DSMLs) [32]. In this chapter, we propose an approach to model-based development of networked cyber-physical systems (CPS) that is centered on the notion of a standardized design specification language. The proposed design specification language can be used to build a CPS design specification model that can serve as a CPS-aware interface between control systems engineer and embedded systems engineer.

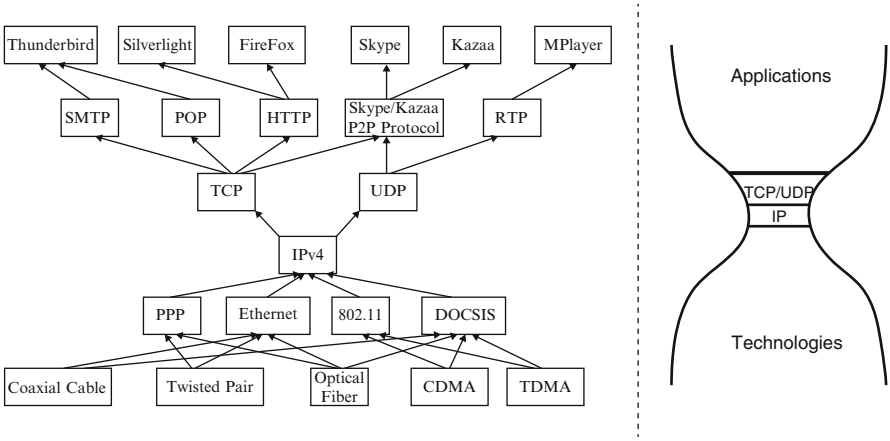

Fig. 3.1 Illustration of hourglass-shaped architecture of Internet; adapted from [2]

The proposed approach is inspired by the hourglass-shaped architecture of Internet, illustrated in Fig. 3.1. The narrow waist of hourglass-shaped architecture suggests that there is less diversity of protocols at this layer of Internet [2]. Any application that can operate based on the services of IP layer can be deployed on the Internet, and any underlying technology that can transport bytes from one point to another according to IP services can be used in the Internet. Similarly, according to the proposed approach to the model-based development of networked CPS (Fig. 3.2), a wide range of DSMLs (and associated analysis tools) can be utilized to develop a platform-imperfection-aware feedback controller design, which is then specified using a standardized CPS design specification language. The proposed feedback controller design can then be analyzed for mapping on to wide range of runtime CPS computing platforms by utilizing their corresponding DSMLs (and associated analysis tools). This approach can support the goals of an integrated CPS theory and development methodology while still taking into account the differences between the domain-specific skillset that control systems engineers and embedded system engineers typically possess.

The rest of the chapter is organized as follows. In Sect. 3.2, we present some related work. In Sect. 3.3, we present the details of the proposed hourglass-shaped architecture for model-based development of networked cyber-physical systems. In Sect. 3.4, we document a number of requirements that any standardized CPS design specification language must satisfy. In Sect. 3.5, we present the overview of a proposed CPS design specification language. In Sects. 3.6–3.8, we discuss the concrete syntax, abstract syntax, and semantics of the proposed CPS design specification language, respectively. In Sect. 3.9, we present the conclusion.

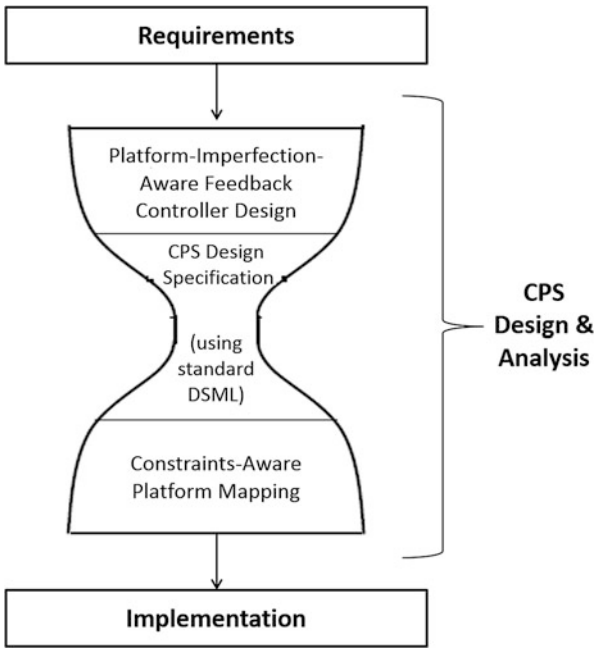

Fig. 3.2 Illustration of hourglass-shaped model of CPS design and analysis process

3.2 Related Work

Figure 3.3 presents a summary of specification languages and analysis tools used in the different stages of a typical embedded control system development process. Simulink [27] (combined with auxiliary tools such as Stateflow [28] and Simscape [26]) has become a de facto standard in the field of embedded control systems for specification and refinement (through simulation) of the feedback controller design, developed by a control engineer through the application of various analytical controller design strategies available in the literature for the field of *control theory* [5]. Once a feedback controller design has shown acceptable performance in the Simulink-based simulation environment, a computer system engineer takes on the task of implementing this feedback controller design as a *real-time computer system*. Various tools have been developed over the years to help a computer systems engineer in this process of converting a feedback controller design from a Simulink-based specification to a real-time computer system implementation. Specialized modeling languages, such as UML (combined with MARTE profile) [30], SysML [10], and AADL [8], help in the process of designing the system and software architecture of the required real-time computer system. Specialized programming languages, such as Lustre [12], Esterel [4], Signal [20], and Giotto [14], help in the development of real-time computer system whose timing performance can be formally guaranteed.

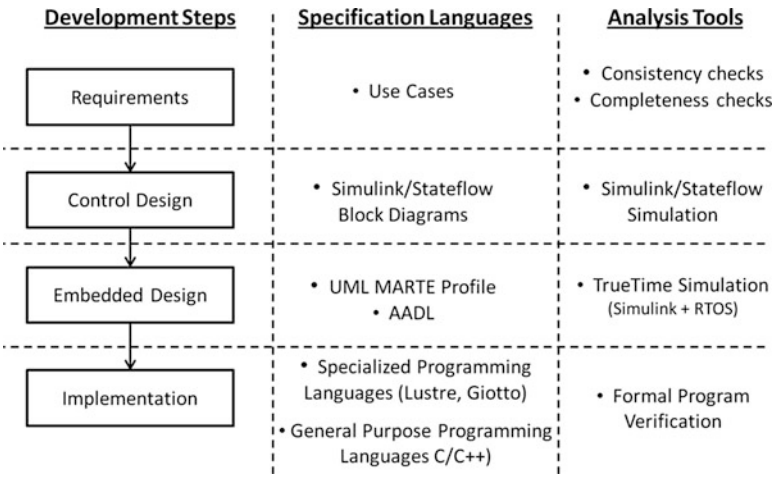

Fig. 3.3 Embedded control systems: development steps, specification languages, and analysis tools; adapted from [36]

Model-based development (MBD) paradigm has also been successfully employed in the domain of embedded control system in order to improve the productivity of a computer systems engineer during the process of conversion of a feedback controller design into a real-time computer system. In MBD paradigm, high-level or platform-independent models (PIM) are transformed into lower-level or platform-specific models (PSM) through the process of model transformation. Both high-level and lower-level models are described using their own domain-specific modeling languages (DSMLs) [32]. A DSML is first defined through a meta-modeling step. A meta-model of a DSML defines the basic constructs (along with their relationships and constraints) that can be used in a DSML. Model transformation step of MBD paradigm uses the meta-models of DSMLs to define transformation rules from higher-level (platform-independent) models to lower-level (platform-specific) models. Model-driven architecture (MDA) [9], model integrated computing (MIC) [18], and eclipse modeling framework (EMF) [11, 33] initiatives represent three popular MBD efforts.

In the domain of embedded control systems, various model transformation (code generation) tools have been developed to automatically generate executable code from Simulink models for various real-time computing platforms. Embedded Coder [25], from Mathworks, Inc., is a commercially available example of such a code generation tool. Another example of a Simulink-based MBD toolset for a more specialized real-time computing platform has been reported in [6].

Building on the MBD paradigm, Sztipanovits et al. [35] describe a methodology for cyber-physical system integration and illustrate their methods on the design of a network of quadrotor UAVs. They identify three design layers: physical, platform, and computation/communication. Their methodology emphasizes component-based design and its associated requirement, compositionality. They identify passivity

as a key characteristic that enables composition of control systems. They identify network characteristics required to compositionally analyze the UAV network.

In a later paper, Sztipanovits et al. [34] describe a CPS methodology and tool suite used for vehicle design. Their tool suite embodies two design platforms: the model integration platform describes the semantic relationships between the models used in design; the tool integration platform describes translations between tools in the flow. Their framework allows them to construct design spaces and analyze the characteristics of those design spaces. Their modeling language CyPhyML includes sublanguages to describe components, system architectures, architectural parameters, analysis models, and testbenches.

However, the CPS model-based development community has not been as successful as some other communities in identifying a design flow which promotes the reuse of tools and can support a range of application domains and implementation targets. For instance, the classic text on compilers [1] identifies several steps in the classical compilation process which are common to a broad class of programming languages: lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, code optimization, and code generation. In this classical compilation process, the intermediate code (developed in an intermediate language such as three-address code) plays a pivotal role by providing an independent narrow interface between a set of source code languages and a set of target machines. Similarly, as illustrated in Fig. 3.1 and detailed in [2], the IP layer can be considered the narrow waist of an hourglass-shaped architecture of Internet. Any application that can operate based on the services of IP layer can be deployed on the Internet, and any underlying technology that can transport bytes from one point to another according to IP services can be used in the Internet.

While model-based development of networked cyber-physical systems is a challenging problem, we believe that abovementioned observations from the domains of software compilation and Internet architecture can be leveraged to improve the model-based development process for networked cyber-physical systems. Therefore, in this chapter, we propose an approach to model-based development of networked cyber-physical systems (CPS) that is centered on the notion of a standardized CPS design specification language, capable of playing an analogous role to the intermediate language and the IP layer from the domains of software compilation and Internet architecture.

3.3 Hourglass-Shaped Architecture for Model-Based CPS Development

Two major categories of CPS research are *platform-imperfection-aware feedback controller design* and *CPS-friendly computing platform design*. Model-based development of cyber-physical systems has the potential to bind the abovementioned CPS research efforts into an integrated, cross-layer CPS development methodology.

This section presents an approach to model-based development of networked cyber-physical systems (CPS) that is centered on the notion of a standardized design specification language. The proposed design specification language can be used to build a CPS design specification model that can serve as a CPS-aware interface between control systems engineer and embedded systems engineer. The proposed approach is inspired by the hourglass-shaped architecture of Internet, illustrated in Fig. 3.1. The narrow waist of hourglass-shaped architecture suggests that there is less diversity of protocols at this middle layer of Internet [2], while many different protocols can be employed at top and bottom layers of Internet.

According to the proposed hourglass-shaped architecture for model-based networked CPS development, illustrated in Fig. 3.2, a wide range of DSMLs (and associated analysis tools) can be utilized to develop a platform-imperfection-aware feedback controller design, which is then specified using a standardized DSML for CPS design specification. Furthermore, according to the proposed hourglass-shaped architecture, the platform-imperfection-aware feedback controller design (specified using the standardized DSML) can then be analyzed for mapping on to various runtime CPS computing platforms by utilizing corresponding DSMLs (and associated analysis tools).

The proposed hourglass-shaped architecture can enable effective coordination between control systems engineer and embedded systems engineer during model-based development of networked cyber-physical system, while still allowing them to concentrate and specialize in the CPS-aware, model-based tools developed in their respective domains. This approach can support the goals of an integrated CPS theory and development methodology while taking into account the differences between the domain-specific skillset that control systems engineer and embedded system engineer must acquire during their respective academic training.

The proposed hourglass-shaped architecture for model-based development of networked CPS consists of three explicit phases: (1) platform-imperfection-aware feedback controller design, (2) CPS design specification, and (3) constraints-aware platform mapping.

3.3.1 Platform-Imperfection-Aware Feedback Controller Design

In this phase, control systems engineer designs a feedback controller that takes into account the imperfections of the runtime computing platform (such as communication delays or failures caused by communication network congestion) at the design time. The resulting “platform-imperfection-aware" feedback controller is either robust against the imperfections of runtime computing platform or possesses the capability to switch between different *control modes* to overcome the imperfections of runtime computing platform. In this phase, control systems engineer utilizes various results from CPS research [37] that have been achieved over the recent

years by leveraging the theoretical advances from the fields of hybrid systems [3], switched systems [21], time-delay systems [7], networked control systems [41], multi-agent networked systems [29], and game theory [16].

During this phase, a control systems engineer can utilize any model-based tool from the following three categories: (a) various DSMLs (and associated analysis tools) that were used in the traditional control system design process [26–28], (b) recently proposed DSMLs (and associated analysis tools) that are employed by the numerous cyber-physical co-design CPS research efforts [13, 31], and (c) any DSMLs (and associated analysis tools) that are proposed by any future CPS research into integrated cyber-physical design.

3.3.2 CPS Design Specification

In this phase, the results of the platform-imperfection-aware feedback controller design process are captured using a standardized DSML for CPS design specification. This CPS design specification must capture the sensed and actuated-upon physical plant parameters as well as the networked controller aspects of a CPS design. However, the networked controller aspects of CPS design should not be described by specifying the runtime computing infrastructure, instead networked controller aspects of CPS design should be described at an abstract level by specifying various *control nodes* and *sensor ports*, *actuator ports*, *input message ports*, and *output message ports* associated with these *control nodes*.

This CPS design specification must also capture the feedback control adaptation strategy to handle the imperfect performance of runtime computing and communication platform. This element of CPS design can also be captured at an abstract level by specifying various *controller modes* of a *control node* and a mode switching logic based on QoS violations associated with *sensor ports*, *actuator ports*, *input message ports*, and *output message ports* of the *control node*. A CPS design specification can also declare some QoS constraints of *sensor ports*, *actuator ports*, *input message ports*, and *output message ports* to be *hard*. This will indicate that these QoS properties must be satisfied by runtime computing platform, because there is no safe backup mode of operation in case of violation of these QoS properties.

3.3.3 Constraints-Aware Platform Mapping

In this phase, the mapping of the CPS design specification (described using standardized DSML) onto various runtime computing platform is analyzed to either choose the most appropriate mapping or figure out the appropriate parameter settings for a runtime computing platform so that the platform can meet the QoS constraints of CPS design (and minimize the time that the system has to spend in a backup mode of operation). During this process, various model transformations can

also be applied to translate the CPS design specification model into appropriate models that can be used as input for corresponding analysis tools (simulation or formal verification) associated with each of the candidate runtime computing platform technologies. Some specialized examples of these runtime computing platforms are Lustre [12], Esterel [4], Signal [20], and Giotto [14] with their own formal computing semantics. More traditional RTOS-based computing platforms can be captured and analyzed through UML (MARTE Profile) or AADL-based models and analysis tools [8, 30].

3.4 Requirements for Standardized CPS Design Specification Language

Following are some of the major requirements that a CPS design specification language (CPS-DSL) must meet:

3.4.1 Physical Plant Parameter Specification

A CPS-DSL must clearly identify the physical plant parameters that are sensed or actuated upon by the feedback controller.

3.4.2 Networked Controller Specification

An appropriate CPS-DSL must also describe the various elements of a networked controller design. These elements include topology of sensors, actuators, and control nodes, local control law for each control node, and information exchanged between different control nodes.

3.4.3 Specification of Controller Adaptation Strategies

For the emerging wide-area CPS application domains, such as smart grid, the performance of communication subsystem cannot be guaranteed. Therefore, CPS-DSL must also define the timing constraints on the information exchange among different control nodes and the control adaptation strategies in case of violation of these timing constraints.

3.4.4 *Interface Between Control Systems Engineer and Real-Time Computer Systems Engineer*

A CPS design specification captures the output of platform-imperfection-aware feedback controller design process, and it also serves as input to the process of developing a functionally equivalent embedded implementation of the feedback controller design. Therefore, the CPS-DSL should be designed in such a way that it can serve as an effective communication interface between control systems engineer and real-time computer systems engineer.

3.4.5 *Formal Semantics*

A CPS design specification language must support formal semantics. The existence of formal semantics of a CPS design specification language (CPS-DSL) opens up the possibility to prove formal equivalence properties between a CPS-DSL-based CPS design specification and the corresponding CPS deployment on a computing platform.

3.5 A Proposed CPS Design Specification Language: Overview

This section presents the summary of a proposed CPS-DSL that can meet the requirements identified in Sect. 3.4. Various aspects (such as concrete syntax, abstract syntax, and semantics) of the definition of proposed CPS-DSL are described in detail in Sects. 3.6–3.8.

The individual language elements of the proposed CPS-DSL can be divided into three categories: *physical system elements*, *cyber system elements*, and *cyber-physical interface elements*. Table 3.1 provides a list of the language elements in each of the abovementioned three categories.

Table 3.1 Language elements of the proposed CPS-DSL

Category	Language elements
Physical system elements	CompoundPhysicalPlant, PhysicalSystemParameter
Cyber-physical interface elements	Sensor, Actuator
Cyber system elements	ComputingNode, CommunicationNetwork, ControlApp, SensorPort, ActuatorPort, InputMsgPort, OutputMsgPort, Mode, ModeSwitchLogic, ControllerFunction, ControllerFunctionMemory, PeriodicControllerInput, PeriodicControllerOutput

3.5.1 Physical System Elements

CompoundPhysicalPlant and *PhysicalSystemParameter* elements belong to the category of *physical system elements*. *CompoundPhysicalPlant* element is used to represent the physical plant of a CPS. A *CompoundPhysicalPlant* element contains a set of *PhysicalSystemParameter* elements. *PhysicalSystemParameter* elements of the proposed CPS-DSL are used to identify the parameters of a physical plant that are to be sensed and actuated upon by the cyber subsystem of a CPS.

3.5.2 Cyber-Physical Interface Elements

Sensor and *Actuator* elements make up the category of *cyber-physical interface elements*. Cyber-physical interface of a CPS design is captured by a set of *Sensor* and *Actuator* elements. Each *Sensor* and *Actuator* element is associated with a corresponding *PhysicalSystemParameter* element.

3.5.3 Cyber System Elements

ComputingNode, *CommunicationNetwork*, *ControlApp*, *SensorPort*, *ActuatorPort*, *InputMsgPort*, *OutputMsgPort*, *Mode*, *ModeSwitchLogic*, *ControllerFunction*, *ControllerFunctionMemory*, *PeriodicControllerInput*, and *PeriodicController-Output* make up the category of *cyber system elements*. Cyber aspects of a CPS design include the topology of computing nodes, the controller application executing on each computing node, and the message exchange among computing nodes. The topology of controller computing nodes is captured by connecting a set of *ComputingNode* elements to a *CommunicationNetwork* element. Each *ComputingNode* element includes a *ControlApp* element and a set of *SensorPort*, *ActuatorPort*, *InputMsgPort*, and *OutputMsgPort* elements. *SensorPort*, *ActuatorPort*, and *ControlApp* elements combine to capture the local control application executing on a computing node.

InputMsgPort and *OutputMsgPort* elements of proposed CPS-DSL are intended to capture the message exchange among computing nodes of a CPS. However, in a generic cyber-physical system, perfect behavior of communication subsystem cannot be guaranteed. As a result, a CPS design must specify the timing constraints on information exchange among computing nodes and different modes of operation for local feedback control law that are used in case of violation of these timing constraints. In the proposed CPS-DSL, *InputMsgPort* and *OutputMsgPort* elements capture the timing constraints on the information exchange among computing node.

Each *ControlApp* element includes a *ModeSwitchLogic* element and a set of *Mode* elements to capture the different modes of operation of feedback control law for handling QoS fault scenarios. Each *Mode* element specifies the control

action taken by the feedback controller in that mode of operation through a set of *ControllerFunction*, *PeriodicControllerInput*, and *PeriodicControllerOutput* elements.

3.6 Proposed CPS Design Specification Language: Concrete Syntax

Since Simulink [27] (combined with auxiliary Stateflow [28] and Simscape [26] blocks) has become a de facto standard in the domain of embedded control systems, concrete syntax of the proposed CPS-DSL has been implemented as an extension to standard blocks available in Simulink. In particular, a new Simulink library [36] has been developed that provides a Simulink block for each element of the proposed CPS-DSL, described in Sect. 3.5. Moreover, Simulink's mask interface capability has been used to provide each new Simulink block with a custom look, and a dialog box for entering element-specific parameters, such as the timing constraints associated with an *InputMsgPort* element.

Figure 3.4 shows a Simulink model that specifies a CPS design using the Simulink-based concrete syntax of the proposed CPS-DSL. Figure 3.5 shows the internal details of a *ComputingNode* block, which contains a *ControlApp* block and a set of *SensorPort*, *ActuatorPort*, *InputMsgPort*, and *OutputMsgPort* blocks. Figure 3.6 shows the internal details of *ControlApp* block, which consists of a set of *Mode* blocks and a *ModeSwitchLogic* block. Figure 3.7 shows the internal details of *Mode* block, which contains a set of *ControllerFunction*, *PeriodicControllerInput*, and *PeriodicControllerOutput* blocks. Figure 3.8 shows the internal details of *ControllerFuncton* block, which contains a description of feedback control law using standard Simulink computation blocks.

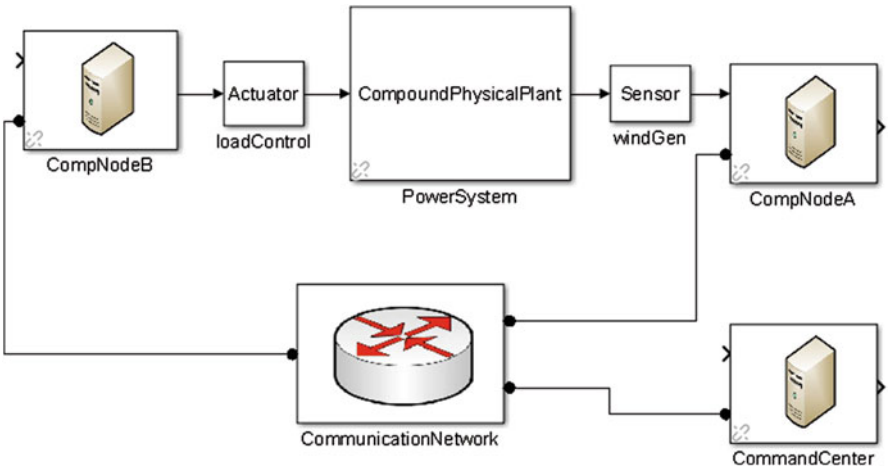

Fig. 3.4 A CPS design, specified as Simulink model with the proposed CPS-DSL

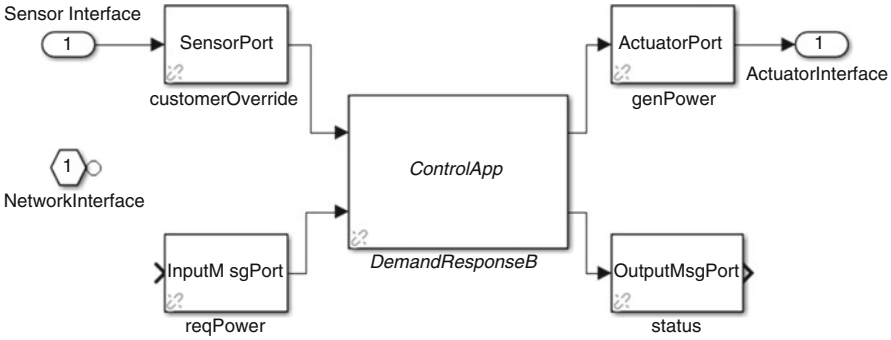

Fig. 3.5 Internal details of *ComputingNode* block, named CompNodeB, in Fig. 3.4

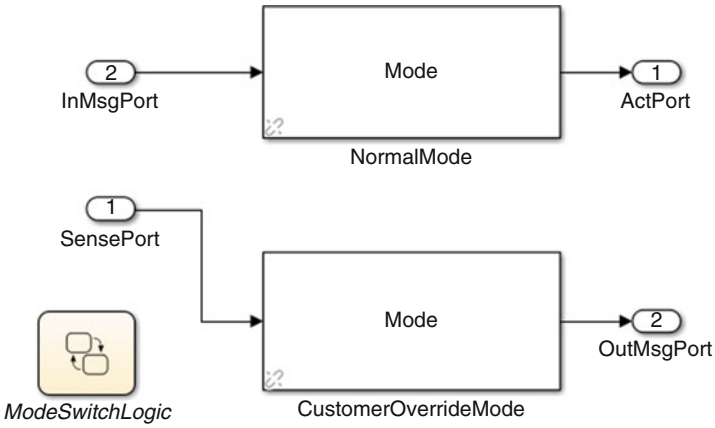

Fig. 3.6 Internal details of *ControlApp* block, named DemandResponseB, in Fig. 3.5

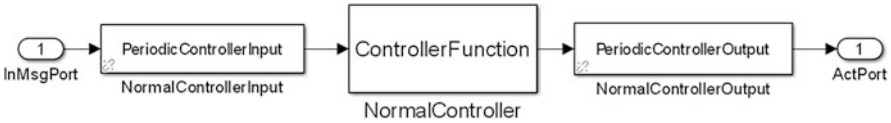

Fig. 3.7 Internal details of *Mode* block, named NormalMode, in Fig. 3.6

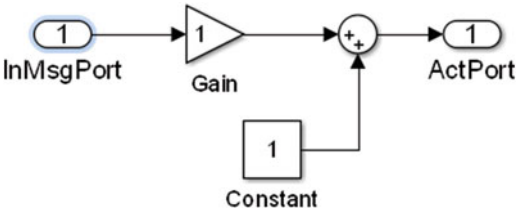

Fig. 3.8 Internal details of *ControllerFunction* block, named NormalControllerFunction, in Fig. 3.7

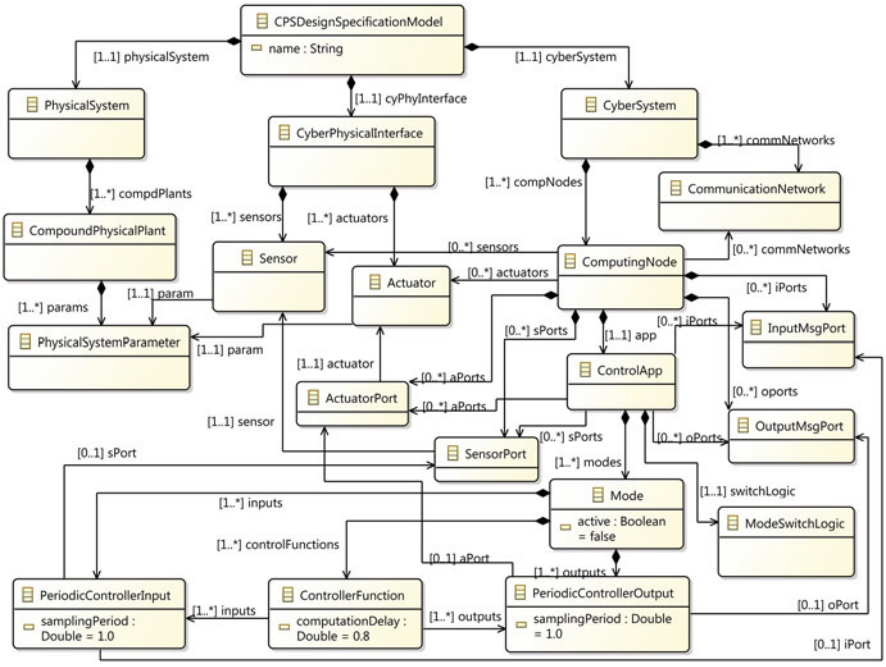

Fig. 3.9 Ecore-based meta-model of proposed CPS-DSL

3.7 Proposed CPS Design Specification Language: Abstract Syntax

Abstract syntax of the proposed CPS-DSL has been implemented as an Ecore-based meta-model [11], combined with a set of object constraint language (OCL)-based constraints. Ecore meta-modeling language was originally developed as a part of Eclipse Modeling Framework (EMF) project [33], while OCL was developed as a part of the UML standardization effort [38]. Figure 3.9 shows a simplified version of the Ecore-based meta-model for the proposed CPS-DSL. Table 3.2 provides some examples of OCL-based constraints that are part of the abstract syntax definition of the proposed CPS-DSL.

3.8 Proposed CPS Design Specification Language: Semantics

According to the semantics of the proposed CPS-DSL, at a given time, only one *Mode* element inside a *ControlApp* is active. *ModeSwitchLogic* element is evaluated at specific time instants, defined by the following two properties of the currently-active mode: mode period and switch frequency from active mode to mode j (the

Table 3.2 Abstract syntax definition of proposed CPS-DSL: examples of OCL-based constraints

context ControlApp inv numOfSimultanoeousActiveModes: modes− >select(active = true)− >size() = 1
context ControllerFunction inv equalityOfSamplingPeriods: inputs− >any(true).samplingPeriod = outputs− >any(true).samplingPeriod
context ControllerFunction inv limitOnComputationDelay: self.computationDelay < outputs− >any(true).samplingPeriod
context CPSDesignSpecificationModel inv noUnusedSensor: cyPhyInterface.sensors− >asSet() = cyberSystem.compNodes.sPorts.sensor− >asSet()
context CPSDesignSpecificationModel inv noUnusedActuator: cyPhyInterface.actuators− >asSet() = cyberSystem.compNodes.aPorts.actuator− >asSet()

number of equally-distant time instants in a single mode period at which the mode switch condition from active mode to mode j is evaluated).

As long as a certain *Mode* element is active, its constituent *PeriodicControllerInput* and *PeriodicControllerOutput* elements periodically sample the values at their inputs and store them at the output until the next sampling time instant. A *ControllerFunction* element contains the specification of feedback control law computation and is always sandwiched between a pair of *PeriodicControllerInput* and *PeriodicControllerOutput* elements with same sampling period T and synchronized sampling instants. The sampling period T, associated with a *ControllerFunction*, is defined in terms of the following two properties: mode period and controller function frequency (the number of equally-distant time instants in a single mode period at which the controller function is evaluated). Moreover, a *ControllerFunction* element takes time Δt to transfer any change in its input to its output where $0 < \Delta t < T$. A *ControllerFunction* element may also contain one or more *ControllerFunctionMemory* elements.

By design, the proposed CPS-DSL leaves its exact semantics dependent on the language used to define the control law computation inside a *ControllerFunction* element. This capability makes the proposed CPS-DSL more flexible. However, for the rest of this chapter, it will be assumed that Simulink computation blocks are used to define the control law computation inside a *ControllerFunction* element.

As outlined in Sect. 3.4.5, semantics of the proposed CPS-DSL should ideally be formally defined. In their seminal work on the application of linear temporal logic (LTL) for formal verification of reactive computer systems, Manna and Pnueli [23, 24] presented a generic model of a reactive computer system in the form of a *transition system*. (This transition system will be referred to as *Manna–Pnueli Transition System* in the rest of this chapter.) They showed that various existing

programming languages and specification formalisms for reactive computer systems can be mapped into this generic model. They also observed that their generic model of reactive computer systems is designed to be capable of capturing any programming language or specification formalism for reactive computer system, proposed in the future. In Sect. 3.8.1, we summarize the abovementioned *Manna–Pnueli Transition System*. In Sect. 3.8.2, we describe the semantics of the proposed CPS-DSL in terms of *Manna–Pnueli Transition System*.

3.8.1 Manna-Pnueli Transition System

Manna–Pnueli Transition System $< \Pi, \Sigma, T, \Theta >$, intended to serve as a generic model for reactive computer systems, consists of the following components:

- $\Pi = \{u_1, \ldots, u_n\}$—A finite set of *state variables*.
  Each state variable is a typed variable, whose *type* indicates the domain from which the values of that variable can be assigned. Some of these state variables are *data variables*, which represent the data elements that are declared and manipulated by the program of a reactive computer system. Other state variables are *control variables*, which keep track of the progress in the execution of a reactive computer system's program.
- Σ—A set of states.
  Each *state* s in Σ is an *interpretation* of Π. An *interpretation* of a set of typed variables is a mapping that assigns to each variable a value in its domain. Therefore, each *state* s in Σ assigns each variable u in Π a value over its domain, which is denoted by $s[u]$.
- T—A finite set of transitions.
  Each transition τ in T represents a state-changing action of the reactive computer system and is defined as a function $\tau : \Sigma \rightarrow 2^{\Sigma}$ that maps a state s in Σ into the (possibly empty) set of states $\tau(s)$ that can be obtained by applying action τ to state s. Each state s' in $\tau(s)$ is defined to be a τ-*successor* of s. A transition τ is said to be *enabled* on s if $\tau(s) \neq \phi$, that is, s has a τ-successor. It is required that one of the transitions, τ_I, called the *idling transition*, is an identity transition, i.e., $\tau_I(s) = \{s\}$ for every state s. The transitions other than the idling transition are called *diligent transitions*.
- Θ—An *initial condition*.
  Initial condition is an assertion (Boolean expression) that characterizes the states at which the execution of reactive computer system's program can begin. A state s satisfying Θ is called an *initial state*.

Each transition τ can be characterized by an assertion $\rho_\tau(\Pi, \Pi')$, called the *transition relation*, of the following form:

$$\rho_\tau(\Pi, \Pi') : C_\tau(\Pi) \wedge (y'_1 = e_1) \wedge \cdots \wedge (y'_k = e_k)$$

This transition relation consists of the following elements:

- An *enabling condition* $C_\tau(\Pi)$, which is an assertion, describing the condition under which the state s may have a τ-successor.
- A conjunction of *modification statements*

$$(y'_1 = e_1) \wedge \cdots \wedge (y'_k = e_k),$$

which relate the values of the state variables in a state s to their values in a successor state s' obtained by applying τ to s. Each modification statement $y_i = e_i$ describes the value of a state variable in state s' as an expression consisting of the state variable values in state s.

As an example, for a transition system with $\Pi = \{x, y, z\}$,

$$\rho_\tau : (x > 0) \wedge (z' = x - y)$$

describes a transition τ that is enabled only when x is positive and this transition assigns the value of z in state s' equal to the value of $x - y$ in state s.

3.8.1.1 Computations

A *computation* of Manna–Pnueli Transition System $< \Pi, \Sigma, T, \Theta >$ is defined to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, \ldots$$

satisfying the following requirements:

- *Initiation*: The first state s_0 is an initial state, i.e., it satisfies the initial condition of the transition system.
- *Consecution*: For each pair of consecutive states s_i, s_{i+1} in σ, $s_{i+1} \in \tau(s_i)$ for some transition τ in T. The pair s_i, s_{i+1} is referred to as a τ-*step*. It is possible for a given pair to be both a τ-step and a τ'-step for $\tau \neq \tau'$.
- *Diligence*: Either the sequence contains infinitely many diligent steps or it contains a terminal state (defined as a state to which only idling transitions can be applied). This requirement excludes the sequences in which, even though some diligent transition is enabled, only idling steps are taken beyond some point. A computation that contains a terminal state is called a *terminating computation*.

Indices i of states in a computation σ are referred to as *positions*. If $\tau(s_i) \neq \phi$ (τ enabled on s_i), it is said that the transition τ is *enabled* at position i of computation σ. If $s_{i+1} \in \tau(s_i)$, it is said that transition τ is *taken* at position i. Several transitions may be enabled at a single position. Moreover, one or more transitions may be considered to be taken at the same position. A state s is called *reachable* in a transition system if it appears in some computation of the system.

3.8.2 Manna–Pnueli Transition System-Based Representation of CPS-DSL

According to the proposed CPS design specification language (CPS-DSL), a *ComputingNode* block contains a *ConrolApp* block and a set of *SensorPort*, *ActuatorPort*, *InputMsgPort*, and *OutputMsgPort* blocks. Furthermore, the *ControlApp* block contains a set of *Mode* blocks and a *ModeSwitchLogic* block. Based on these constituent blocks, a *ComputingNode* block, $CompNode1$, of CPS-DSL can be represented as the Manna–Pnueli Transition System, $P_{CompNode} < \Pi_{P_{CompNode}}, \Sigma_{P_{CompNode}}, T_{P_{CompNode}}, \Theta_{P_{CompNode}} >$, outlined below, where:

- $\Pi_{P_{CompNode}}$—A finite set of *state variables*.

$$
\begin{aligned}
\Pi_{P_{CompNode1}} = \{ & t, t^{switch}_{CompNode1}, mode_{CompNode1}, t^{next}_{CompNode1}, \\
& sensePort^{1}_{CompNode1}, sensePort^{2}_{CompNode1}, \\
& \quad \ldots, sensePort^{p}_{CompNode1}, \\
& inMsgPort^{1}_{CompNode1}, inMsgPort^{2}_{CompNode1}, \\
& \quad \ldots, inMsgPort^{r}_{CompNode1}, \\
& actPort^{1}_{CompNode1}, actPort^{2}_{CompNode1}, \ldots, actPort^{q}_{CompNode1}, \\
& outMsgPort^{1}_{CompNode1}, outMsgPort^{2}_{CompNode1}, \\
& \quad \ldots, outMsgPort^{l}_{CompNode1}, \\
& periodicControllerIn^{1}_{CompNode1}, \\
& periodicControllerIn^{2}_{CompNode1}, \\
& \quad \ldots, periodicControllerIn^{a}_{CompNode1}, \\
& periodicControllerOut^{1}_{CompNode1}, \\
& periodicControllerOut^{2}_{CompNode1}, \\
& \quad \ldots, periodicControllerOut^{b}_{CompNode1}, \\
& controllerFunctionMemory^{1}_{CompNode1}, \\
& controllerFunctionMemory^{2}_{CompNode1}, \\
& \quad \ldots, controllerFunctionMemory^{c}_{CompNode1} \}
\end{aligned}
$$

where

t = time,
$t^{switch}_{CompNode1}$ = latest mode switch time of *ControlApp* block, associated with *ComputingNode* block $CompNode1$,
$mode_{CompNode1}$ = current mode of *ControlApp* block, associated with *ComputingNode* block $CompNode1$,
$t^{next}_{CompNode1}$ = next relevant time instant (actuator update, output message update) during the current mode of operation of *ControlApp* block, associated with *ComputingNode* block $CompNode1$,
$sensePort^{i}_{CompNode1}$ = A *SensorPort* block, contained in the *ComputingNode* block $CompNode1$,
$inMsgPort^{i}_{CompNode1}$ = An *InputMsgPort* block, contained in the *ComputingNode* block $CompNode1$,
$actPort^{i}_{CompNode1}$ = An *ActuatorPort* block, contained in the *ComputingNode* block $CompNode1$,,
$outMsgPort^{i}_{CompNode1}$ = An *OutputMsgPort* block, contained in the *ComputingNode* block $CompNode1$,
$peridoicControllerIn^{i}_{CompNode1}$ = A *PeriodicControllerInput* block that is contained in a mode of the *ControlApp* block, associated with *ComputingNode* block $CompNode1$,
$peridoicControllerOut^{i}_{CompNode1}$ = A *PeriodicControllerOutput* block that is contained in a mode of the *ControlApp* block, associated with *ComputingNode* block $CompNode1$,
$controllerFunctionMemory^{i}_{CompNode1}$ = A *ControllerFunctionMemory* block that is contained in the *ControllerFuction* block of a mode of the *ControlApp* block, associated with *ComputingNode* block $CompNode1$,

- $\Sigma_{P_{CompNode}}$—A set of states.
  Each *state* s in Σ is an *interpretation* of Π. An *interpretation* of a set of typed variables is a mapping that assigns to each variable a value in its domain. The domain of state variables t, $t^{switch}_{CompNode1}$, and $t^{next}_{CompNode1}$ is $\mathbf{R}_{\geq 0}$. The domain of state variable $mode_{CompNode1}$ is $Modes_{CompNode1}$ = {Set of modes of *ControlApp* block, contained in the *ComputingNode* block $CompNode1$}. Given the following definitions of Π_α and $\mathbf{D}$, all the state variables in Π_α have the domain $\mathbf{D}$:

$$\Pi_\alpha = \{sensePort^{i}_{CompNode1}, actPort^{i}_{CompNode1}, outMsgPort^{i}_{CompNode1},$$
$$periodicControllerIn^{i}_{CompNode1}, periodicControllerOut^{i}_{CompNode1},$$
$$controllerFunctionMemory^{i}_{CompNode1}\}$$

$$\mathbf{D} = \{x \mid (x \in \mathbf{R})$$
$$\wedge\ (x \text{ can be represented by type } double \text{ of computer system})\}$$

The state variable $inMsgPort^{i}_{CompNode1}$ has the following domain:

$$\mathbf{P} = \{(x, y) \mid (x \in \mathbf{R}) \wedge (y \in \mathbf{D})\}$$

- $T_{P_{CompNode}}$—A finite set of transitions.

$$T_{P_{CompNode1}} = \tau_I \cup T^{ModeSwitches}_{CompNode1} \cup T^{TimeIncrement}_{CompNode1}$$

where

τ_I = Idling Transition

$T^{ModeSwitches}_{CompNode1}$ = $\{\tau^{mode_i mode_j}_{CompNode1} \mid \exists$ a *mode switch* from $mode_i$ to $mode_j$ in the *ModeSwitchLogic* block of *ControlApp* block, associated with *ComputingNode* block $CompNode1\}$

$T^{TimeIncrement}_{CompNode1}$ = $\{\tau^{mode_1}_{CompNode1}, \tau^{mode_2}_{CompNode1}, \ldots, \tau^{mode_M}_{CompNode1}\}$

As outlined in the summary of Manna–Pnueli Transition System approach, presented in Sect. 3.8.1, each transition τ can be characterized by an *enabling condition* and a *set of modification statements*. Based on the abovementioned set of transitions $T_{P_{CompNode1}}$ of $P_{CompNode1}$, all the diligent transitions of $P_{CompNode1}$ can be completely described through the enabling conditions and modification statements of the following generic transitions: $\tau^{mode_i mode_j}_{CompNode1}$ and $\tau^{mode_i}_{CompNode1}$.

(a) $\tau^{mode_i mode_j}_{CompNode1}$*: Enabling Condition*

$$C_{\tau^{mode_i mode_j}_{CompNode1}} = (mode_{CompNode1} == mode_i) \wedge ModeSwitchCondition_{CompNode1}(t, mode_i, mode_j) \wedge ModeSwitchCheckTime_{CompNode1}(t, t^{switch}_{CompNode1}, mode_i, mode_j)$$

where

$ModeSwitchCondition_{CompNode1}(t, mode_i, mode_j)$ = An assertion that returns true if the *mode switch condition* associated with *mode switch* from $mode_i$ to $mode_j$ in the $ModeSwitchLogic$ block, contained in the *ComputingNode* block $CompNode1$, is true at time t.

$ModeSwitchCheckTime_{CompNode1}(t, t^{switch}_{CompNode1}, mode_i, mode_j)$ = An assertion that returns true if $t - t^{switch}_{CompNode1} = a\{\frac{Period_{mode_i}}{SwitchFreq_{mode_i mode_j}}\}$, for some $a \in \{1, 2, \ldots, SwitchFreq_{mode_i mode_j}\}$.

(b) $\tau^{mode_i mode_j}_{CompNode1}$*: Modification Statements*

1. $mode_{CompNode1}{}' = mode_j$
2. $t^{switch}_{CompNode1}{}' = t$
3. $t^{next}_{CompNode1}{}' = t + t_{jump}$
   where

$$t_{jump} = \min \Big\{ t_j \mid (t_j > 0) \wedge (t + t_j = t^{switch}_{CompNode1}{}' + a\{\frac{Period_{mode_j}}{ControllerFunctionFreq_{controllerFucntion_d}}\}),$$

for some

$$a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_d}\}$$

and for some

$$controllerFunction_d \in ControllerFunctions^{mode_j}_{CompNode1} \Big\}$$

4.

$$periodicControllerOuts^{mode_j}_{CompNode1}{}' = ModeSwitchFunction^{mode_i mode_j}_{CompNode1}(periodicControllerOuts^{mode_i}_{CompNode1})$$

where
$ModeSwitchFunction^{mode_i mode_j}_{CompNode1}$ = A function that produces the values to which $periodicControllerOuts^{mode_j}_{CompNode1}$ are initialized after the *mode switch* from $mode_i$ to $mode_j$ of *ControlApp*, associated with $CompNode1$

5.

$$actPorts^{mode_j}_{CompNode1}{}' = ControllerOutsToActs^{mode_j}_{CompNode1}(periodicControllerOuts^{mode_j}_{CompNode1}{}')$$

where
$ControllerOutsToActs^{mode_j}_{CompNode1}$ = A function that captures the input–output relationship (produced by the combined effect) of all the connections between *PeriodicControllerOutput* blocks and *ActuatorPort* blocks in $mode_j$ of $CompNode1$.

6.

$$outMsgPorts^{mode_j}_{CompNode1}{}' = ControllerOutsToOutMsgs^{mode_j}_{CompNode1}(periodicControllerOuts^{mode_j}_{CompNode1}{}')$$

where

$ControllerOutsToOutMsgs^{mode_j}_{CompNode1}$ = A function that captures the input–output relationship (produced by the combined effect) of all the connections between *PeriodicControllerOutput* blocks and *OutputMsgPort* blocks in $mode_j$ of $CompNode1$.

7.

$$periodicControllerIns_{controllerFucntion_b}{}'$$
$$= LoadControllerInputs^{mode_j}_{controllerFunction_b}(sensePorts^{mode_j}_{CompNode1}{}', inMsgPorts^{mode_j}_{CompNode1}{}', periodicControllerOuts^{mode_j}_{CompNode1}{}')$$

for every $controllerFunction_b \in ControllerFunctions^{mode_j}_{CompNode1}$

where

$LoadControllerInputs^{mode_j}_{controllerFunction_b}$ = A function that captures the input–output relationship (produced by the combined effect) of all the connections between *PeriodicControllerInput* blocks, associated with *ControllerFunction* block $controllerFunction_b$ in $mode_j$, and *SensorPorts, InputMsgPorts,* and *PeriodicControllerOutput* blocks in $mode_j$ of $CompNode1$.

(c) $\tau^{mode_i}_{CompNode1}$*: Enabling Condition*

$$C_{\tau^{mode_i}_{CompNode1}} = (mode_{CompNode1} == mode_i) \wedge \neg(ModeSwitchCondition_{CompNode1}(t, mode_i, mode_c) \wedge ModeSwitchCheckTime_{CompNode1}(t, t^{switch}_{CompNode1}, mode_i, mode_c))$$

$$\forall mode_c \in \{mode_c \mid \exists\ a\ mode\ switch\ from\ modei\ to\ modec\ of\ ControlApp\ associated\ with\ ComputingNode\ block\ CompNode1\}$$

(d) $\tau^{mode_i}_{CompNode1}$*: Modification Statements*

1. $t' = t^{next}_{CompNode1}$
2. $t^{next}_{CompNode1}{}' = t' + t_{jump}$

where

$$t_{jump} = \min\left\{t_j \mid (t_j > 0) \wedge (t' + t_j = t^{switch}_{CompNode1} + a\{\frac{Period_{mode_i}}{ControllerFucntionFreq_{controllerFunction_d}}\})\right.$$

for some $a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_d}\}$
and
for some $controllerFunction_d \in ControllerFunctions^{mode_i}_{CompNode1}$ }

3.

$$(periodicControllerOuts_{controllerFunction_e}{}', controllerFunctionMemory_{controllerFunction_e}{}') = f^{controllerFunction_e}(periodicControllerIns_{controllerFunction_e}, controllerFuctionMemory_{controllerFunction_e})$$

$$\forall controllerFunction_e \in \{controllerFunction_e \mid (controllerFunction_e \in ControllerFunctions^{mode_i}_{CompNode1}) \wedge (t' = t^{switch}_{CompNode1} + a\{\frac{Period_{mode_i}}{ControllerFunctionFreq_{controllerFunction_e}}\})$$

for some $a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_e}\}\}$

where
$f^{controllerFunction_e}$ = The function implemented by the internal components (Simulink blocks) of *ControllerFunction* block *controller Fucntion*$_e$.

4.

$$periodicControllerIns_{controllerFunction_f}{}' = LoadControllerInputs^{mode_i}_{controllerFunction_f}(sensePorts^{mode_i}_{CompNode1}{}', inMsgPorts^{mode_i}_{CompNode1}{}', periodicControllerOuts^{mode_i}_{CompNode1}{}')$$

$$\forall controllerFunction_f \in \{controllerFunction_f \mid (controllerFunction_f \in ControllerFunctions^{mode_i}_{CompNode1}) \wedge (t' = t^{switch}_{CompNode1} + a\{\frac{Period_{mode_i}}{ControllerFunctionFreq_{controllerFunction_f}}\})$$

for some $a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_f}\}\}$

5.

$$actPorts_{CompNode1}^{mode_i}{}' = ControllerOutsToActs_{CompNode1}^{mode_i}(periodicControllerOuts_{CompNode1}^{mode_i}{}')$$

6.

$$outMsgPorts_{CompNode1}^{mode_i}{}' = ControllerOutsToOutMsgs_{CompNode1}^{mode_i}(periodicControllerOuts_{CompNode1}^{mode_i}{}')$$

- $\Theta_{P_{CompNode}}$—An *initial condition*. Any initial state s of transition system $P_{CompNode}$ must satisfy the following initial conditions:

$t = 0$
$t_{CompNode1}^{switch} = 0$
$mode_{CompNode1} = mode_1$
$t_{CompNode1}^{next} = \min \Big\{ t_j \mid (t_j > 0) \wedge (t_j = a\{\frac{Period_{mode_1}}{ControllerFunctionFreq_{controllerFunction_d}}\})$ for some $a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_d}\}$ and for some $controllerFunction_d \in ControllerFunctions_{CompNode1}^{mode_1} \Big\}$

3.9 Conclusion

Taking inspiration from the hourglass-shaped architecture of the Internet, this chapter has proposed an hourglass-shaped architecture for model-based development of networked cyber-physical systems. Similar to the central role played by TCP/IP protocols in the Internet architecture, the proposed architecture for model-based networked CPS development is centered on the notion of a standardized CPS design specification language.

The proposed hourglass-shaped architecture can enable effective coordination between control systems engineers and embedded systems engineers during a model-based CPS development process, while still acknowledging the differences between the domain-specific skillset that control systems engineer and embedded system engineer typically possess. The chapter has also proposed a version of the abovementioned CPS design specification language and discussed its various aspects such as concrete syntax, abstract syntax, and semantics.

References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Boston: Addison-Wesley Longman Publishing Co., Inc.
2. Akhshabi, S., & Dovrolis, C. (2013). The evolution of layered protocol stacks leads to an hourglass-shaped architecture. In *Dynamics on and of complex networks* (Vol. 2, pp. 55–88). New York: Springer.
3. Antsaklis, P. J. (1998). Hybrid control systems: An introductory discussion to the special issue. *IEEE Transactions on Automatic Control, 43*(4), 457–460.
4. Berry, G., & Gonthier, G. (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming, 19*(2), 87–152.
5. Brogan, W. L. (1991). *Modern control theory*. Upper Saddle River: Prentice-Hall.
6. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., & Niebert, P. (2003). From simulink to scade/lustre to TTA: A layered approach for distributed embedded applications. In *ACM sigplan notices* (Vol. 38, pp. 153–162). New York: ACM.
7. Dugard, L., & Verriet, E. (1998). *Stability and control of time-delay systems*. *Lecture notes in control and information sciences*. Berlin: Springer.
8. Feiler, P. H., & Gluch, D. P. (2012). *Model-based engineering with AADL: An introduction to the SAE architecture analysis & design language*. Boston: Addison-Wesley.
9. Frankel, D. S. (2003). *Model driven architecture: Applying MDA to enterprise computing*. Hoboken: Wiley.
10. Friedenthal, S., Moore, A., & Steiner, R. (2014). *A practical guide to SysML: The systems modeling language*. Burlington: Morgan Kaufmann.
11. Gronback, R. C. (2009). *Eclipse modeling project: A domain-specific language toolkit*. Boston: Addison-Wesley Professional.
12. Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous data flow programming language lustre. *Proceedings of the IEEE, 79*(9), 1305–1320.
13. Henriksson, D., & Elmqvist, H. (2011). Cyber-physical systems modeling and simulation with Modelica. In *International Modelica Conference* (Vol. 9). Linköping: Modelica Association.
14. Henzinger, T., Horowitz, B., & Kirsch, C. (2003). Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE, 91*(1), 84–99.
15. Henzinger, T. A., & Sifakis, J. (2006). The embedded systems design challenge. In *FM 2006: Formal Methods* (pp. 1–15). Berlin: Springer.
16. Jones, M., Kotsalis, G., & Shamma, J. S. (2013). Cyber-attack forecast modeling and complexity reduction using a game-theoretic framework. In *Control of cyber-physical systems* (pp. 65–84). Heidelberg: Springer.
17. Kang, W., Kapitanova, K., & Son, S. H. (2012). Rdds: a real-time data distribution service for cyber-physical systems. *IEEE Transactions on Industrial Informatics, 8*(2), 393–405.
18. Karsai, G., Sztipanovits, J., Ledeczi, A., & Bapty, T. (2003). Model-integrated development of embedded software. *Proceedings of the IEEE, 91*(1), 145–164.
19. Lee, E. A. (2009). Computing needs time. *Communications of the ACM, 52*(5), 70–79. https://doi.org/10.1145/1506409.1506426
20. LeGuernic, P., Gautier, T., Le Borgne, M., & Le Maire, C. (1991). Programming real-time applications with signal. *Proceedings of the IEEE, 79*(9), 1321–1336.
21. Liberzon, D., & Morse, A. S. (1999). Basic problems in stability and design of switched systems. *IEEE Control Systems, 19*(5), 59–70.
22. Liu, I., Reineke, J., Broman, D., Zimmer, M., & Lee, E. A. (2012). A PRET microarchitecture implementation with repeatable timing and competitive performance. In *IEEE 30th International Conference on Computer Design (ICCD), 2012* (pp. 87–93). https://doi.org/10.1109/ICCD.2012.6378622
23. Manna, Z., & Pnueli, A. (1991). *The temporal logic of reactive and concurrent systems: Specification*. New York: Springer.

24. Manna, Z., & Pnueli, A. (1995). *Temporal verification of reactive systems: Safety*. New York: Springer.
25. Mathworks inc. (2016). Embedded coder r2015b. http://www.mathworks.com/products/embedded-coder/
26. Mathworks inc. (2016). Simscape r2015b. http://www.mathworks.com/products/simscape/
27. Mathworks inc. (2016). Simulink r2015b. http://www.mathworks.com/products/simulink/
28. Mathworks inc. (2016). Stateflow r2015b. http://www.mathworks.com/products/stateflow/
29. Mesbahi, M., & Egerstedt, M. (2010). *Graph theoretic methods in multiagent networks*. Princeton: Princeton University Press.
30. Selic, B., & Gérard, S. (2013). *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. New York: Elsevier.
31. Simko, G., Lindecker, D., Levendovszky, T., Neema, S., & Sztipanovits, J. (2013). Specification of cyber-physical components with formal semantics–integration and composition. In *Model-driven engineering languages and systems* (pp. 471–487). Berlin: Springer.
32. Stahl, T., Völter, M., Bettin, J., Haase, A., & Helsen, S. (2006). *Model-driven software development: Technology, engineering, management*. Hoboken: Wiley.
33. Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse modeling framework*. Boston: Addison-Wesley Professional.
34. Sztipanovits, J., Bapty, T., Koutsoukos, X., Lattmann, Z., Neema, S., & Jackson, E. (2018). Model and tool integration platforms for cyber-physical system design. *Proceedings of the IEEE, 106*, 1–26. https://doi.org/10.1109/JPROC.2018.2838530
35. Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., & Gupta, V. (2012). Toward a science of cyber–physical system integration. *Proceedings of the IEEE, 100*(1), 29–44. https://doi.org/10.1109/JPROC.2011.2161529
36. Tariq, M. U., Florence, J., & Wolf, M. (2014). Design specification of cyber-physical systems: Towards a domain-specific modeling language based on simulink, eclipse modeling framework, and giotto. In: *ACESMB@ MoDELS* (pp. 6–15).
37. Tarraf, D. C. (2013). Control of cyber-physical systems. In *Proceedings of Lecture Notes in Control and Information Sciences* (Vol. 449).
38. Warmer, J. B., & Kleppe, A. G. (2003). The object constraint language: Getting your models ready for MDA. Boston: Addison-Wesley Professional.
39. Wolf, W. (2009). Cyber-physical systems. *Computer, 42*(3), 88–89. https://doi.org/10.1109/MC.2009.81
40. Wolf, M., & Serpanos, D. (2017). Safety and security of cyber-physical and internet of things systems [point of view]. *Proceedings of the IEEE, 105*(6), 983–984. https://doi.org/10.1109/JPROC.2017.2699401
41. Zhang, W., Branicky, M. S., & Phillips, S. M. (2001). Stability of networked control systems. *IEEE Control Systems, 21*(1), 84–99.