SERVICE-ORIENTED REFERENCE MODEL FOR CYBER-PHYSICAL SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Muhammad Umer Tariq

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2016

Copyright © 2016 by Muhammad Umer Tariq

SERVICE-ORIENTED REFERENCE MODEL FOR CYBER-PHYSICAL SYSTEMS

Approved by:

Dr. Marilyn Wolf, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Santiago Grijalva
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Magnus Egerstedt
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Eric Feron
School of Aerospace Engineering
Georgia Institute of Technology

Date Approved: April 18, 2016

To my parents (Farzana Tariq and Tariq Rashid) who sacrificed immensely to provide me with opportunities in life, but still allowed me the freedom to make my own choices in life.

ACKNOWLEDGEMENTS

First of all, I would like to acknowledge the efforts of all the people who have contributed over the years into making Georgia Tech the great institution that it is today, because the diverse opportunities I have been provided in the PhD program at Georgia Tech have enabled me to expand my horizons and grow into a person with clarity about professional and personal goals for the rest of my life. I also wish to acknowledge various sponsors of my graduate studies: Fulbright Foreign Student Program, Higher Education Commission Pakistan, National Science Foundation (NSF), and Advanced Research Projects Agency for Energy (ARPA-E).

My deepest gratitude goes to my Ph.D. advisor Dr. Marilyn Wolf who afforded me the freedom to discover my strengths and interests, introduced me to interesting research literature, guided me in formulating and solving relevant research problems, and provided encouragement and support throughout the process. I also want to thank Dr. Santiago Grijalva and Dr. Magnus Egerstedt who provided invaluable guidance and showed incredible patience that allowed me to tackle the inter-disciplinary nature of the field of cyber-physical systems. Special thanks also go to Dr. George Riley and Dr. Eric Feron for serving on my dissertation defense committee. Their time investment is very much appreciated.

I have a debt of gratitude to Jacques Florence, Hasan Nasir, Dr. Mohammad Abdullah Al Faruque, Arun Padmanabhan, and Dr. Brian Swenson for being such excellent collaborators in our joint efforts towards research publications. During my PhD research, I also had the opportunity to collaborate with many members of ACES Lab and GRITS Lab on the ARPA-E GENI project. It has been a pleasure to work with all of them and learn from them. I am also deeply thankful to Dr. Abubakr

Muhammad for not only providing me an exciting summer research position, but also being a source of inspiration during my doctoral studies.

I am grateful to many friends, especially Bilal, Hassan, Wasim, Wasif, Abdul Majid, Minhaj, and Abdul Qadir, whose delightful company (and patience for my endless rants about the game of cricket) made it possible to deal with the stress of living away from family for long periods of time. I can never repay the debt of gratitude that I owe to my friend and mentor Mohammad Omer for not only helping me during my early days at Georgia Tech but also providing a role model for living a positive, simplified, and relaxed existence that I always hope to emulate.

I am also deeply grateful to my parents (Farzana Tariq and Tariq Rashid) and my siblings (Amara and Kamran) for their endless support during the seemingly endless period of my doctoral studies. Finally, i must thank my lucky stars for being blessed with the love, support, and companionship of my wife (Sarwat), who is an unwavering source of strength for me in all my endeavors in life.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . **iv**

LIST OF TABLES . . . . **xi**

LIST OF FIGURES . . . . **xii**

SUMMARY . . . . **xv**

I INTRODUCTION **1**

II LITERATURE SURVEY **9**

2.1 Reference Model 9

2.2 Real-Time Computer Systems 10

2.3 Embedded Control Systems 12

2.4 Cyber-Physical Systems 14

2.5 Service-Oriented Computing 18

III SERVICE-ORIENTED REFERENCE MODEL FOR CYBER PHYSICAL SYSTEMS **20**

3.1 Development Steps 21

3.1.1 Requirements Engineering 22

3.1.2 Platform-aware Feedback Controller Design 22

3.1.3 CPS Design Specification 22

3.1.4 CPS Design Refinement through Simulation 22

3.1.5 Service-based Decomposition of CPS Design 23

3.1.6 Service Publication and Discovery 23

3.1.7 Resource-aware Service Deployment 23

3.1.8 QoS-aware Service Operation 24

3.1.9 Service Update 24

3.2 Formal Performance Guarantees 24

3.3 Advantages over Task-based Reference Model 25

IV TECHNOLOGICAL REQUIREMENTS OF SERVICE-ORIENTED REFERENCE MODEL FOR CYBER-PHYSICAL SYSTEMS . 27

4.1 CPS Design Specification Language 28

4.2 Simulation Environment for CPS Design Refinement 29

4.3 Service Description Language 29

4.3.1 Service Interface 29

4.3.2 Service Resources 29

4.3.3 Service Modes 30

4.4 Service-based Computing Platform for CPS Computing Nodes . . . 30

4.5 Automated Model Transformation between CPS Design Specification and Service Descriptions 31

V CASE STUDY: SMART GRID 32

5.1 Demand Response Application 33

5.2 Power Agreement Application 34

5.3 Incremental Co-deployment of Smart Grid Applications 37

VI CPS DESIGN SPECIFICATION LANGUAGE 39

6.1 Requirements 39

6.1.1 Physical Plant Specification 39

6.1.2 Networked Controller Specification 39

6.1.3 Specification of Controller Adaptation Strategies 40

6.1.4 Interface between Control Engineer and Real-time Computer Systems Engineer 40

6.2 Design 41

6.2.1 Language Elements 41

6.2.2 Concrete Syntax 43

6.2.3 Abstract Syntax 45

6.2.4 Semantics 45

6.3 Case Study 46

VII SIMULATION ENVIRONMENT FOR CPS DESIGN REFINEMENT . . . 47

7.1 Requirements . . . 47

7.1.1 Co-simulation of Physical and Cyber Subsystems . . . 47

7.1.2 Simulation of Computer Networks . . . 48

7.1.3 Simulation of Application-level Software . . . 48

7.1.4 Automated Configuration of Simulation Engine . . . 49

7.2 Design . . . 49

7.3 Case Study . . . 52

VIII SERVICE DESCRIPTION LANGUAGE . . . 57

8.1 Requirements . . . 57

8.1.1 Service Interface . . . 57

8.1.2 Service Resources . . . 57

8.1.3 Service Modes . . . 58

8.2 Design . . . 58

8.2.1 Giotto Programming Language . . . 59

8.2.2 Extensions to Giotto Programming Language . . . 61

8.3 Case Study . . . 62

IX SERVICE-BASED COMPUTING PLATFORM . . . 65

9.1 Requirements . . . 65

9.2 Design . . . 66

9.2.1 Embedded Machine (E Machine) . . . 66

9.2.2 Combination of Embedded Machine (E Machine) and Compiler Machine (C Machine) . . . 68

9.3 Case Study . . . 69

X FORMAL PERFORMANCE GUARANTEES . . . 71

10.1 Formal Methods: A Short Introduction . . . 71

10.2 Formal Methods for Reactive Computer Systems: Manna-Pnueli Approach . . . 72

10.2.1 Manna-Pnueli Transition System 73

10.2.2 Computations 75

10.2.3 Behavioral Equivalence 76

10.3 Proposed Extensions to Manna-Pnueli Approach 78

10.4 Manna-Pnueli Transition System Representation: CPS Computing Node in CPS-DSL 80

10.5 Manna-Pnueli Transition System Representation: CPS Computing Node with 1 CPS Service 80

10.6 Manna-Pnueli Transition System Representation: CPS Computing Node with k CPS Services 81

10.7 Functional Equivalence of CPS Design Specification and Service-based CPS Deployment 81

10.8 Non-interference between Co-deployed CPS Services 97

XI SIMULATION-BASED SMART GRID TESTBEDS: DEMONSTRATING THE ADVANTAGES OF SERVICE-ORIENTED CPS REFERENCE MODEL 106

11.1 Smart Grid Testbed: Traditional Service-Oriented Computing 106

11.2 Smart Grid Testbed: Proposed Service-Oriented CPS Approach . . . 108

11.3 Smart Grid Case Study: Demonstration of the Advantages of Proposed Service-Oriented CPS Approach 110

11.3.1 Smart Grid Case Study: Task-based Embedded Control Systems Approach 110

11.3.2 Smart Grid Case Study: Traditional Enterprise-Domain Service-Oriented Computing Approach 112

11.3.3 Smart Grid Case Study: Proposed Service-Oriented CPS Approach 112

XII CONCLUSION 115

12.1 Summary of Contributions 116

12.2 Future Directions 117

APPENDIX A — MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE IN CPS-DSL . . . 119

APPENDIX B — MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE WITH 1 CPS SERVICE 127

APPENDIX C — MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE WITH K CPS SERVICES 133

REFERENCES 156

VITA 157

LIST OF TABLES

5.1 Division of IEEE 24-bus Case into 10 Prosumers 36

7.1 Co-Simulation Procedure 52

8.1 Service Description of *DemandResponseServiceB* using the Proposed Giotto-based Service Description Language 64

LIST OF FIGURES

1.1 Task-based reference model for embedded control systems. 3

1.2 Service-oriented reference model for cyber-physical systems. 5

2.1 Real-time computer system as a part of embedded control system. . . 12

2.2 A summary of state-of-the-art approach and tools for development of embedded control systems. 13

2.3 Overview of service-oriented computing in the domain of enterprise system integration; adapted from [16]. 18

3.1 Service-oriented reference model for cyber-physical systems. 21

3.2 Incremental deployment of applications: task-based reference model for embedded control systems (Notice the disruption in system operation during period $[t4, t5]$). 25

3.3 Incremental deployment of applications: service-oriented CPS reference model. 25

4.1 Technological requirements of a service-oriented reference model for cyber-physical systems. 28

5.1 IEEE 24-bus case [28]. 33

5.2 Demand response application. 34

5.3 Prosumer network graph. 35

5.4 Summary of power agreement control law [62]. 37

5.5 Computing nodes for a prosumer network. 38

6.1 Role of the CPS Design Specification Language (presented in this chapter) in the service-oriented reference model for cyber-physical systems. 40

6.2 A CPS design, specified as Simulink model with the proposed CPS-DSL. 43

6.3 Internal details of *ComputingNode* block, named CompNodeB, in Figure 6.2. 44

6.4 Internal details of *ControlApp* block, named DemandResponseB, in Figure 6.3. 44

6.5 Internal details of *Mode* block, named NormalMode, in Figure 6.4. . . 44

6.6 Internal details of *ControllerFunction* block, named NormalControllerFunction, in Figure 6.5. 44

6.7 Ecore-based meta-model of proposed CPS-DSL. 45

7.1 Role of the simulation environment for CPS design refinement (presented in this chapter) in the service-oriented CPS reference model. . 48

7.2 Structure of the proposed simulation environment for CPS design refinement. 50

7.3 Additions to the standard structure of ns-3 network simulator. 50

7.4 UML class diagram for proposed simulation environment. 51

7.5 Demand response application: power generation and consumption profiles for different link delays. 53

7.6 Demand response application: power generation and consumption profiles for different controller update periods. 53

7.7 Demand response application: power generation and consumption profiles under communication network congestion. 54

7.8 Power agreement application: convergence behavior under two different combinations of communication link delay and controller update period (only 3 out of 10 prosumers are depicted for readability). . . . 55

7.9 Power agreement application: convergence behavior under network congestion on point-to-point communication links to ProsumerCompNode for prousmer1 (only 4 out of 10 prosumers are depicted for readability). 55

8.1 Role of the CPS Service Description Language (presented in this chapter) in the service-oriented CPS reference model. 58

8.2 Typical configuration of Giotto and E Machine for embedded control systems. 60

8.3 Major programming elements of Giotto language. Proposed extensions for a Giotto-based CPS service description language are shown in red with dotted lines. 62

8.4 Graphical representation of service descriptions for *DemandResponseServiceB*. 63

9.1 Role of the service-based computing platform (presented in this chapter) in the service-oriented CPS reference model. 66

9.2 Typical configuration of Giotto and E Machine for embedded control systems. 67

9.3 Proposed solution for the requirement of a service-based computing platform. 69

9.4 Case study: demand response application with proposed service-based computing platform. 70

10.1 Basic framework employed by the field of formal methods. 72

10.2 Formal methods for reactive computer systems: Manna-Pnueli approach. 73

11.1 Structure of the simulation-based smart grid testbed with traditional enterprise-domain, service-oriented computing paradigm. 107

11.2 Additions to the standard structure of ns-3 network simulator as a component of smart grid testbed with traditional SOC paradigm. . . 107

11.3 Structure of the simulation-based smart grid testbed with proposed CPS-enabled, service-oriented computing paradigm. 109

11.4 Additions to the standard structure of ns-3 network simulator as a component of smart grid testbed with the proposed CPS-enabled SOC paradigm. 109

11.5 Demand response application from case study in Chapter 5: performance comparison before and after the deployment of the power agreement application (resource overloading on ProsumerCompNode computing platforms for prousmer1 and prosumer10). 111

11.6 Power agreement application from case study in Chapter 5: convergence behavior under resource overloading on ProsumerCompNode computing platforms for prousmer1 and prosumer10 (only 4 out of 10 prosumers are depicted for readability). 111

11.7 Demand response application from case study in Chapter 5: performance after the successful deployment of CPS services associated with power agreement application. 113

11.8 Power agreement application from case study in Chapter 5: convergence behavior after successful field deployment through CPS services on a computing infrastructure that was already supporting a demand response application (only 4 out of 10 prosumers are depicted for readability). 113

SUMMARY

This dissertation has formalized a service-oriented computing (SOC) based approach to cyber-physical systems (CPS) in the form of a *service-oriented CPS reference model*. The proposed reference model extends the traditional SOC paradigm for handling hard real-time CPS aspects by introducing resource-aware service deployment and quality-of-service (QoS)-aware service operation phases alongwith the mandate for following formal guarantees: 1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and 2) non-interference between the co-deployed CPS services from the perspective of their timing performance. As a result, the proposed CPS reference model enables a provably-correct process for converting a new CPS application from a CPS design specification to a service-based CPS deployment in the field without affecting the timing performance of already deployed CPS applications or disrupting the operation of already deployed CPS applications for system upgrade. Therefore, unlike the traditional task-based reference model from the domains of automotive and avionics, the proposed service-oriented CPS reference model enables disruption-free incremental system deployment and reconfiguration that are fundamental requirements of the emerging safety-critical but large scale and "always-online" CPS application domains such as smart grid and vehicular networks.

Although the development of suitable technologies for a domain according to the requirements of a reference model for that domain is meant to be an on-going effort by a research community, this dissertation has contributed to this effort by proposing solutions for the following technological requirements of service-oriented CPS reference model: 1) CPS design specification language, 2) simulation environment for

CPS design refinement, 3) service description language, and 4) service-based computing platform for CPS computing nodes. By leveraging the Manna-Pnueli approach of formal methods for reactive computer systems, this dissertation has also shown how the aforementioned technological solutions combine to provide the formal performance guarantees, mandated by the proposed CPS reference model. Finally, this dissertation has also presented simulation-based smart grid testbeds that can be used to demonstrate the advantages of the proposed service-oriented CPS approach in a virtual environment before its implementation on safety-critical, live smart grid infrastructure.

CHAPTER I

INTRODUCTION

During the age of industrialization, the human race conquered many physical processes of the universe and used them for its own advantage. These achievements were enabled by the field of *feedback control systems*, which deals with the process of controlling a physical system through a feedback controller [7]. Traditionally, these feedback controllers were implemented in the analog domain using different electric circuit elements. However, the advent of computation and networking technologies created the opportunity to implement these feedback controllers more easily and flexibly in the digital domain as a special breed of computer systems, known as a *real-time computer systems*, which are characterized by the need to perform computations under timing constraints. The resulting configuration of a feedback control system, in which the feedback controller is implemented as a real-time computer system, is referred to as *embedded control system* [3]. Some prime examples of embedded control systems are automotive and avionics systems [61] [12].

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 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.

To facilitate the development process of embedded control systems, various tools and technologies have been developed by different stakeholders, over the years, in a somewhat isolated and ad-hoc manner. However, the relationship and integration of these tools and technologies can be studied by utilizing the concept of a *reference model*. A reference model for a domain is defined as an ontology, consisting of a set of interlinked and unifying concepts for that domain. A reference model is designed to enable clear communication among various stakeholder of the domain as well as the development of a coherent and consistent set of technologies and tools for that domain [58] [63]. For the domain of embedded control systems, a "task-based reference model" has been proposed in the literature [45]. According to this task-based reference model, an embedded control system can be described by three elements:

1. *Controller application model* that describes the feedback control algorithm as a set of *tasks*. Each task is a unit of computation that needs to be done by the feedback controller.

2. *Computing platform model* that describes the available computing platform as a set of *processors* and *resources*. *Processors* are active entities such as central processing units, transmission links, and database servers, while *resources* are passive entities such as memory, mutexes, and database locks.

3. Set of *task scheduling algorithms*. Each task must have one or more *processors* and *resources* in order to make progress on its assigned unit of computation. When a task has the required *processors* and *resources*, it is said to be "scheduled" and it can "execute" its unit of computation at a certain speed.

According to this task-based reference model, major steps in the development of embedded control system are requirements engineering, feedback controller design, controller design refinement through simulation, task-based feedback controller specification, task implementation, task priority assignment, task deployment, and testing (or formal verification). Figure 1.1 summarizes the major elements and development methodology of the task-based reference model for embedded control systems.

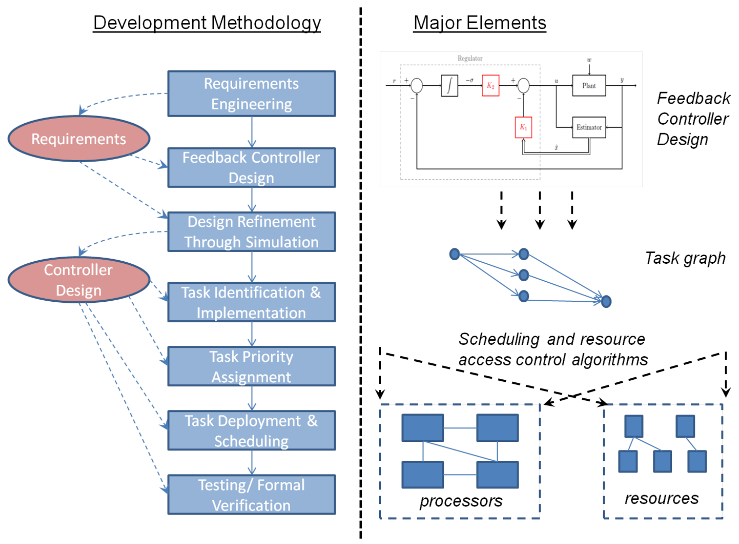

Development Methodology
Requirements Engineering
Requirements
Feedback Controller Design
Design Refinement Through Simulation
Controller Design
Task Identification & Implementation
Task Priority Assignment
Task Deployment & Scheduling
Testing/ Formal Verification
Major Elements
Feedback Controller Design
Task graph
Scheduling and resource access control algorithms
processors
resources

Figure 1.1: Task-based reference model for embedded control systems.

Based on the above mentioned summary of task-based reference model, it can be seen that various state-of-the-art tools and technologies in the domain of embedded control systems have evolved into a form that is consistent with this reference model. For instance, real-time operating systems support task deployment with different task priorities and provide various task scheduling algorithms [66]. General purpose programming languages as well as specialized programming languages for the embedded control system domain (such as Giotto [25]) provide a task-based programming model. Formal analysis tools have been developed that study the schedulability of multiple time-constrained tasks on a computing node [10]. Various code generation tools have been developed that automatically translate a Simulink-based description of feedback controller into task-based source code [51] [50].

Dramatic decrease in the cost of communication and computation technologies, seen in the last two decades, has enabled the development of a new breed of embedded control systems that are much larger in scale such as smart grid [75], vehicular networks [54], and automated irrigation networks [72]. Besides their larger scale, this new breed of embedded control systems have other distinguishing characteristics such as their "always-online" nature and a much longer lifecycle. Reliable development of this new breed of embedded control systems through traditional development tools, which were based on a task-based reference model, will result in unsustainable development and maintenance costs, because these traditional tools are ill-equipped to provide appropriate support for disruption-free incremental system deployment and system reconfiguration that are fundamental requirements for handling the larger-scale, "always-online" nature, and longer life-cycles of this new breed of systems.

Over the last few years, limitations of traditional embedded control system development techniques have spawned the new field of *cyber-physical systems* (CPS), which takes a fresh look at the abstractions used in the traditional embedded control system development process. CPS research aims 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 integrated CPS theory and development toolset will enable the reliable development and maintenance of more complex versions of traditional embedded control systems (such as automotive and avionics) as well as the emerging larger scale and "always online" embedded control systems (such as smart grid and vehicular networks) with manageable costs.

However, advances in CPS research still focus on the traditional task-based programming model of a real-time computer system, historically popular in the automotive and avionics domains. As noted earlier, the task-based model is a relatively low-level of abstraction for a real-time computer system and provides poor support for disruption-free incremental system deployment and reconfiguration. As a result, these advances in CPS research, by themselves, cannot effectively handle the unique challenges posed by the larger scale and "always online" nature of emerging CPS application domains such as smart grid and vehicular networks.

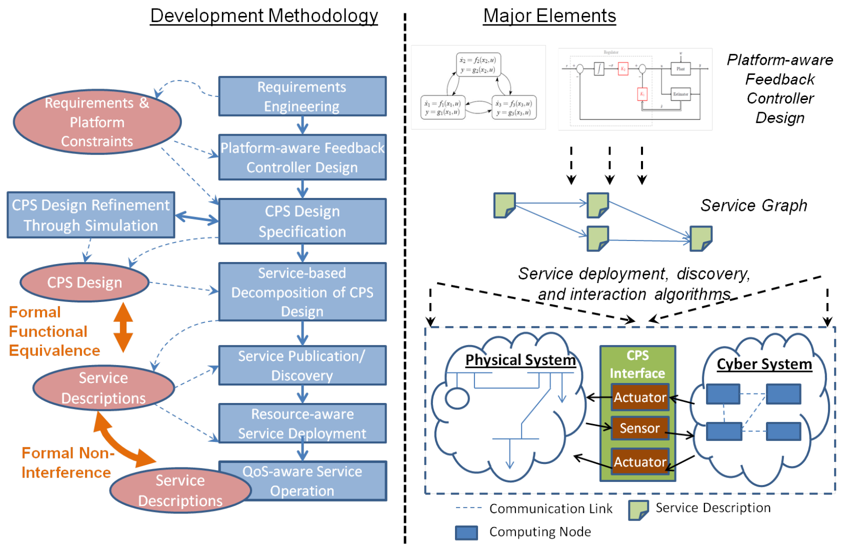

Development Methodology
Major Elements
Requirements & Platform Constraints
Requirements Engineering
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
QoS-aware Service Operation
Service Descriptions
Platform-aware Feedback Controller Design
Service Graph
Service deployment, discovery, and interaction algorithms
Physical System
CPS Interface
Actuator
Sensor
Actuator
Cyber System
Communication Link
Service Description
Computing Node

Figure 1.2: Service-oriented reference model for cyber-physical systems.

This dissertation has formalized a service-oriented computing (SOC) based approach to cyber-physical systems (CPS) in the form of a *service-oriented CPS reference model*. SOC paradigm can inherently provide support for disruption-free incremental system deployment and reconfiguration, required for handling the larger scale, "always-online" nature, and longer lifecycle of above mentioned emerging CPS application domains such as smart grid. However, the proposed reference model also extends the traditional SOC paradigm for handling hard real-time CPS aspects by introducing resource-aware service deployment and quality-of-service (QoS)-aware service operation phases with certain formal performance guarantees. According to the proposed reference model, each CPS scenario is described by three elements:

1. *CPS application model* that describes the CPS application to be supported by the system as a set of resource- and QoS-aware service descriptions.

2. *CPS platform model* that describes the available CPS platform as a set of computing nodes, communication links, sensors, actuators, and physical system entities.

3. Set of algorithms that achieve resource-aware service deployment and QoS-aware

service operation.

According to the proposed service-oriented reference model, major steps in CPS development are requirements engineering, platform-aware feedback controller design, CPS design specification, CPS design refinement through simulation, service-based decomposition of CPS design, service publication and discovery, resource-aware service deployment, QoS-aware service operation, and service update. The proposed reference model also requires the existence of formal guarantees for the following aspects: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and (2) non-interference between the co-deployed CPS services from the perspective of their timing performance. The existence of these formal guarantees enables a provably-correct process for converting a new CPS application from a CPS design specification to a service-based CPS deployment in the field without affecting the timing performance of already deployed CPS applications.

By adopting the proposed service-oriented CPS reference model, CPS development effort can focus on the platform-aware feedback controller design and simulation-based design refinement. Once the performance of a CPS design has been found to be satisfactory in these two steps, the CPS design can be transformed into a service-based field deployment in an automated and provably-correct manner, without worrying about its effects on the existing applications supported by the same CPS computing platform. As a result, unlike the task-based reference model, the proposed service-oriented CPS reference model enables disruption-free incremental system deployment and reconfiguration of emerging safety-critical but large scale and "always online" CPS application domains such as smart grid and vehicular networks.

Figure 1.2 summarizes the major elements and development methodology of the proposed service-oriented CPS reference model. This dissertation also identifies some important technological requirements that must be met to enable CPS development and operation based on the proposed reference model. Furthermore, this dissertation presents solutions for the following technological requirements of the proposed CPS reference model: CPS design specification language, simulation environment for CPS design refinement, service-description language, and service-based computing platform for CPS computing nodes. By extending and applying the Manna-Pnueli Approach [47] of formal methods for reactive computer systems, this dissertation also presents formal proofs that show the capability of aforementioned technological solutions to provide the following guarantees, mandated by the proposed reference model: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS deployment and (2) non-interference between the co-deployed services from the perspective of their timing performance. Finally, this dissertation also presents simulation-based smart grid testbeds that can be used to demonstrate the advantages of the proposed service-oriented CPS approach in a virtual environment before its implementation on safety-critical, live smart grid infrastructure.

The structure of this dissertation is as follows. Chapter 2 reviews some relevant research literature. Chapter 3 outlines the proposed service-oriented CPS reference model. Chapter 4 identifies the technological requirements that must be met in order to enable CPS development according to the proposed service-oriented reference model. Chapter 5 presents a smart grid case study that is used in the following chapters to explain various elements of the proposed technological solutions. Next four chapters (Chapter 6, Chapter 7, Chapter 8, and Chapter 9) of the dissertation present solutions for the following technological requirements of the proposed CPS reference model: CPS design specification language, simulation environment for CPS design refinement, service-description language, and service-based computing platform for CPS computing nodes. Chapter 10 shows how the proposed technological solutions provide the formal performance guarantees, required by the service-oriented CPS reference model. Using the smart grid case study from Chapter 5, Chapter 11 presents a performance comparison of task-based embedded control systems approach, enterprise-domain service-oriented computing approach and the proposed service-oriented CPS approach through simulation-based smart grid testbeds. Finally, Chapter 12 summarizes the novel contributions made through this research and delivers concluding remarks.

CHAPTER II

LITERATURE SURVEY

The literature survey, presented in this chapter, reviews some relevant previous research in the realm of reference models, real-time computer systems, embedded control systems, cyber-physical systems, and service-oriented computing.

2.1 Reference Model

A reference model for a domain is an abstract conceptual framework, consisting of a small number of interlinked and unifying concepts for that domain. A reference model is designed to enable clear communication about the domain among various stakeholders. A reference model is not a standard or implementation technology in itself. However, it does "inform" the development of a set of compatible standards and technologies for a certain domain [58] [9].

In the past, the concept of a reference model has been successfully employed in various domains to enable the development of a coherent set of technologies and standards for that domain. Following are some examples of reference models, developed for various domains:

- Open Systems Interconnection (OSI) Reference Model for communication systems [83]

- Agent Systems Reference Model (ASRM) for multi-agent systems [63]

- National Institute of Standards and Technology (NIST) Reference Model for software engineering environments [9]

- National Institute of Standards and Technology (NIST) Reference Model for project support environments [8]

- Task-based Reference Model for real-time computer systems [45]

Similarly, the development of an appropriate reference model for cyber-physical systems (CPS) can not only ensure clear communication among different stakeholders, but also help in the process of developing a coherent and consistent set of standards and technologies for cyber-physical systems. However, any reference model for cyber-physical systems must be based on concepts that are generic enough to be reconciled with existing technologies (such as Simulink-based controller design refinement [51] and various industry-standard real-time operating systems and time-sensitive middleware products [39] [40]), but still provide valuable guidance for the evolution of existing standards and technologies into a consistent and coherent set of future standards and technologies.

2.2 Real-Time Computer Systems

In the context of computer systems engineering, a *real-time computer system* is a computer system which must respond as quickly as required by the users of the computer system or as necessitated by the process being controlled by that computer system [49] [45]. The field of real-time computer systems engineering has various facets such as computing platforms for real-time systems, application development for real-time systems, model-driven development of real-time systems, and performance analysis of real-time systems [40] [39].

The computing platform for a real-time system typically consists of some computing hardware accompanied by some variation of a real-time operating system (RTOS). An overview of architectures and principles employed in real-time operating systems is presented in [66]. A *task* is a logical abstraction of a program that is schedulable by an RTOS. A *task* is represented by a data structure containing an identity, priority, state of execution, and resources allocated to the task. An RTOS performs three important functions related to *tasks*: scheduling, dispatching, and inter-task communication and synchronization.

Real-time computer system applications are typically developed by using the task-based programming model provided by an RTOS. However, in the recent past, the subject of model-driven development (MDD) has received considerable attention due to its potential for improving the software development productivity [65]. In MDD paradigm, high-level or platform-independent models (PIM) are transformed into lower-level or platform-specific models (PSM) through the process of model transformation. High-level models are typically created using a domain-specific modeling language (DSML). The syntax of DSML and lower-level platform is defined in a meta-modeling step. A meta-model defines the basic constructs that can be used in a modeling language. Model transformation step typically uses the meta-models of DSML and the platform to define transformation rules from high-level models to low-level platform specific code. Model-Driven Architecture (MDA) [19], Model Integrated Computing (MIC) [38], and Eclipse Modeling Framework (EMF) [67] [23] initiatives represent three popular MDD efforts. However, it must be noted that the current MDD toolsets for real-time computer systems employ task-based programming model, provided by RTOS, as the low-level platform model.

Performance analysis of a typical computer system is usually carried out in the testing phase of a software development process. However, real-time computer systems are frequently employed in safety-critical applications. Therefore, it is not sufficient to "show" (through testing) that the system does not have errors. In many cases, real-time system developers must "prove" that the system does not have errors [2] [56]. As a result, a lot of research has been focused on techniques that allow system designers to estimate, predict, or prove the performance of a real-time computer system at an early stage in the development process. The task-based model of real-time computer system has been used to formalize this performance analysis issue

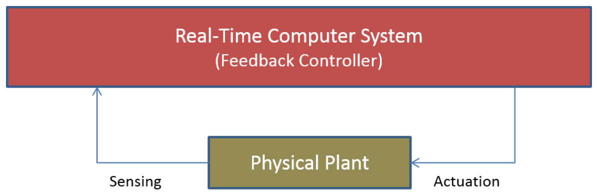


Figure 2.1: Real-time computer system as a part of embedded control system.

as a scheduling theory problem, and various useful results have been obtained over the years [10] [45].

2.3 Embedded Control Systems

The field of *feedback control systems* deals with the process of controlling a physical plant through a feedback controller. Traditionally, these feedback controllers were implemented in the analog domain using different electric circuit elements. However, the advent of computing and networking technologies created the opportunity to implement these feedback controllers more easily and flexibly in the digital domain as a *real-time computer system*. The resulting configuration of a feedback control system (shown in Figure 2.1), in which the feedback controller is implemented as a real-time computer system, is referred to as *embedded control system* [3]. Some prime examples of embedded control systems are automotive and avionics systems [61] [12].

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 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. To facilitate the development process of embedded control systems, various tools and technologies have been developed by different stakeholders over the years. Figure 2.2 presents a summary of specification languages and analysis tools used in the different

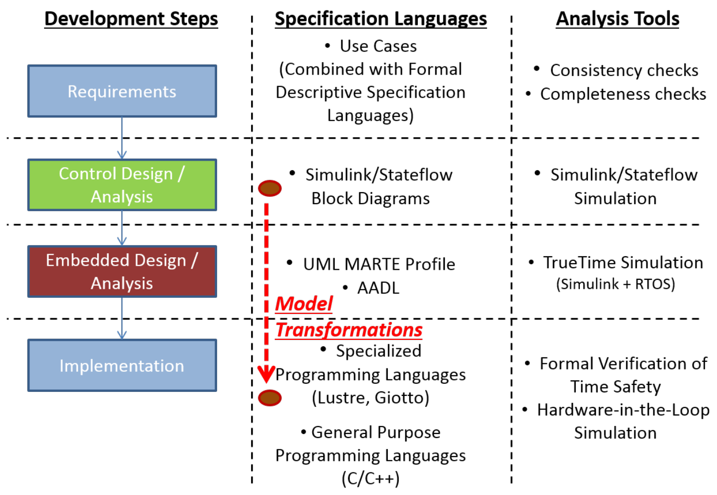

Development Steps
Specification Languages
Analysis Tools
Requirements
• Use Cases (Combined with Formal Descriptive Specification Languages)
• Consistency checks
• Completeness checks
Control Design / Analysis
• Simulink/Stateflow Block Diagrams
• Simulink/Stateflow Simulation
Embedded Design / Analysis
• UML MARTE Profile
• AADL
• TrueTime Simulation (Simulink + RTOS)
Model Transformations
Implementation
• Specialized Programming Languages (Lustre, Giotto)
• General Purpose Programming Languages (C/C++)
• Formal Verification of Time Safety
• Hardware-in-the-Loop Simulation

Figure 2.2: A summary of state-of-the-art approach and tools for development of embedded control systems.

stages of a typical embedded control system development process.

Simulink, developed by MathWorks, Inc., is a simulation and model-based design tool that provides a graphical editor for specifying a model as a set of hierarchical block diagrams [51]. Simulink is often used in conjunction with some auxiliary tools that provide specialized types of blocks to be used in Simulink block diagram. Two important examples of such auxiliary tools are Stateflow [52] and Simscape [29]. Stateflow allows the users to model decision logic based on the state machine and flow chart formalisms. Simscape provides fundamental building blocks from various domains (such as electrical, mechanical, and hydraulic) that can be combined to model a physical plant. Simulink (combined with auxiliary tools such as Stateflow and Simscape) has become a defacto 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* [7] [59].

Once a feedback controller design has shown acceptable performance in the Simulink-based simulation environment, a computer system engineer takes on the 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 system 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) [64], SysML [20], and AADL [17], help in the process of designing the system and software architecture of the required real-time computer system. Specialized programming languages, such as Lustre [24], Esterel [6], and Signal [42], help in the development of real-time computer system whose timing performance can be formally guaranteed. However, it must be noticed that the above mentioned modeling languages as well as programming languages work with the assumption of a task-based programming model for real-time computer system that requires the re-implementation and testing of the whole real-time computer system if the same computing platform is used at a later stage (of system upgrade or reconfiguration) to support the real-time implementation of another feedback controller.

Model-driven development (MDD) has also been successfully employed in the domain of embedded control system in order to improve the productivity of a computer system engineer during the process of conversion of a feedback controller design into a real-time computer system. 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 [50], from Mathworks, Inc., is a commercially-available example of such a code generation tool. Another example of a Simulink-based MDD toolset for a more specialized real-time computing platform has been reported in [11].

2.4 Cyber-Physical Systems

As detailed in the last section, the field of *embedded control systems* brings together the fields of *control theory* and *real-time computer systems*. However, as noted in [27], the fields of *control theory* and *real-time computer systems* employ two completely different types of models: analytical models and computational models. As a result, two very different design processes are used in the two stages of embedded control system development process: feedback controller design and feedback controller implementation as real-time computer system. These inherent differences have resulted in a set of development methodologies for embedded control systems, which 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 operating conditions. As a result, these development methodologies provide poor support for system upgrade and reconfiguration, because any small change in the system requirements and design creates the need to take the system offline and repeat the expensive system testing process. Therefore, traditional development techniques for embedded control systems are not capable of efficiently handling the ever increasing complexity of traditional applications (such as automotive and avionics) and larger scale and "always-online" nature of emerging applications (such as smart grid, vehicular networks, and automated irrigation networks).

These limitation 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) [79]. 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 integrated CPS theory and development toolset will enable the reliable development and maintenance of more complex versions of traditional embedded control systems (such as automotive and avionics) as well as the emerging larger scale and "always online" embedded control systems (such as smart grid) with manageable costs.

By leveraging the theoretical developments from the fields of hybrid systems [4], switched systems [15] [43], time-delay systems [14], networked control systems [82], multi-agent networked systems [53], and game theory [33], CPS research has focused on a "platform-aware" feedback controller design process for embedded control system applications [74]. This controller design process 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. The resulting "platform-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.

CPS research has also proposed specialized computing platforms that have more predictable timing performance. Some examples of this approach are provided in [44], [36], and [41]. Co-design of control and real-time computing aspects of a systems has also been addressed by CPS research, as seen in [81]. Furthermore, CPS researchers have addressed the issue of converting a high-level controller model to a provably-correct implementation as the source code of a real-time computing platform. For instance, this issue is addressed in [32] by converting model-level theoretical properties, such as stability and convergence, into code-level assertions and invariants for C code. The need for an integrated CPS development toolset has also been the focus of considerable research effort as demonstrated by numerous initiatives towards analytic virtual integration [48] and model-driven development(MDD) [37] for cyber-physical systems.

These advances in CPS research still focus on the traditional task-based programming model of a real-time computer system, historically popular in the automotive and avionics domains. As noted earlier, the task-based model is a relatively low-level of abstraction for a real-time computer system and provides poor support for disruption-free incremental system deployment and reconfiguration. As a result, the above mentioned CPS solutions, by themselves, cannot effectively handle the unique challenges posed by the larger scale and "always online" nature of emerging CPS application domains such as smart grid.

Building on the CPS research, summarized above, this dissertation has formalized a service-oriented computing (SOC) [21] based approach to cyber-physical systems in the form of a *reference model*. The proposed CPS reference model advocates the use of a CPS design specification language (CPS-DSL) to capture the results of the above mentioned "platform-aware" feedback controller design process. According to the proposed reference model, this CPS design specification serves as input for the processes of CPS design refinement through cyber-physical co-simulation and the field deployment of a service-based CPS application. The proposed CPS reference model also requires the existence of formal guarantees for the following aspects: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and (2) non-interference between the co-deployed CPS services from the perspective of their timing performance. The existence of these formal guarantees will provide a provably-correct process for converting a new CPS application from a CPS design specification to a service-based CPS deployment in the field without affecting the performance of already deployed CPS applications.

By adopting the proposed service-oriented CPS reference model, CPS development effort can focus on the platform-aware feedback controller design and simulation-based design refinement. Once the performance of a CPS design has been found to be satisfactory in these two steps, the CPS design can be transformed into a service-based field deployment in an automated and provably-correct manner, without worrying about its effects on the existing applications supported by the same CPS computing platform. As a result, unlike the task-based reference model, the proposed service-oriented CPS reference model and associated technological solutions will enable the

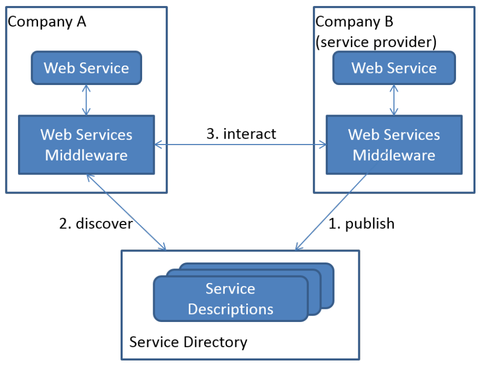


Figure 2.3: Overview of service-oriented computing in the domain of enterprise system integration; adapted from [16].

disruption-free incremental system deployment and reconfiguration that are fundamental requirements of the emerging safety-critical but large scale and "always online" CPS application domains such as smart grid and vehicular networks.

2.5 Service-Oriented Computing

Because of its potential for developing flexible systems, service-oriented computing (SOC) paradigm has seen an increase in its popularity over the last decade. In the SOC paradigm, software applications take one of the following three roles: service consumers, service brokers and service producers. Service producers publish their services to service brokers (service directories) by using their service descriptions. Service consumers discover these services by contacting the service brokers. Once service consumers have discovered these services, they directly interact with services, hosted by service-producers, through the exchange of messages. Thus, three major aspects of SOC paradigm are service description, service discovery and service interaction. Efforts to standardize these aspects have resulted in Web Services, a set of standards that deal with these three major aspects of service-oriented computing [16].

As illustrated in Figure 2.3, the SOC paradigm has traditionally been used for enterprise integration applications. However, recent efforts in the fields of service-oriented system engineering (SOSE) [76] and device profile for web services (DPWS) [31] have tried to move SOC concepts from enterprise application domain to the embedded computing domain. The focus of these efforts has been the interoperability of networked embedded devices. These efforts have not concentrated on enhancing the traditional SOC paradigm with mechanisms that will allow its application to the complete range of real-time systems, especially those with hard timing constraints. This research tries to address these concerns by adding resource-aware service deployment and quality-of-service (QoS)-aware service operation phases to the traditional SOC paradigm [68] [69]. These developments make service-oriented computing a good candidate for serving as the foundation of a generic CPS reference model.

CHAPTER III

SERVICE-ORIENTED REFERENCE MODEL FOR CYBER PHYSICAL SYSTEMS

This chapter presents the details of the proposed reference model for cyber-physical systems (CPS). The proposed reference model is based on the service-oriented computing (SOC) paradigm [21], because this paradigm is uniquely suitable for handling the larger scale, "always-online" nature, and longer life-cycles of emerging CPS application domains such as smart grid, vehicular networks, and automated irrigation networks. Currently, SOC paradigm is being used widely in the enterprise computing domain through Web Services technology [16]. However, the traditional SOC paradigm cannot be directly applied to the domain of cyber-physical systems, because it is not capable of handling the hard real-time aspects of cyber-physical systems. To address this limitation of the traditional SOC paradigm, the proposed CPS reference model extends the traditional SOC paradigm by introducing resource-aware service deployment and QoS-aware service operation phases with certain formal performance guarantees.

According to the proposed reference model, each CPS scenario is described by three elements:

1. A *CPS application model* that describes the CPS application to be supported by the system as a set of resource- and QoS-aware service descriptions.

2. A *CPS platform model* that describes the available CPS platform as a set of computing nodes, communication links, sensors, actuators, and physical system entities.

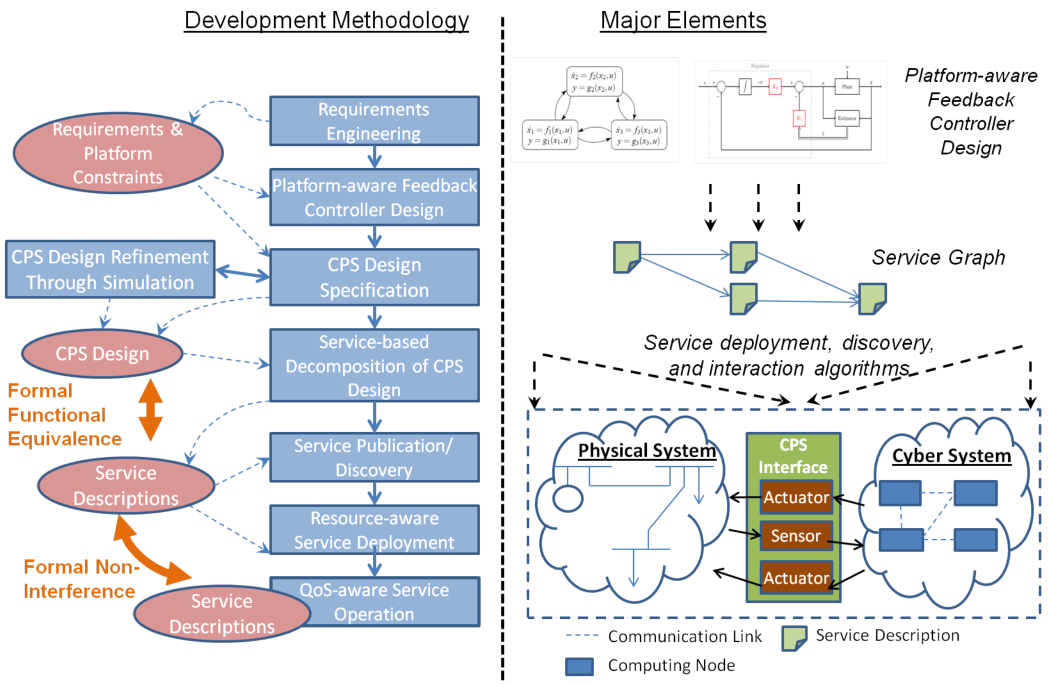

Development Methodology
Major Elements
Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
Service Descriptions
QoS-aware Service Operation
Platform-aware Feedback Controller Design
Service Graph
Service deployment, discovery, and interaction algorithms
Physical System
CPS Interface
Actuator
Sensor
Actuator
Cyber System
Communication Link
Service Description
Computing Node

Figure 3.1: Service-oriented reference model for cyber-physical systems.

3. A set of algorithms that achieve resource-aware service deployment and QoS-aware service operation.

Figure 3.1 shows three major elements of the proposed service-oriented reference model for cyber-physical systems.

3.1 Development Steps

As shown in Figure 3.1, major development steps for a cyber-physical system, according to the proposed reference model, are requirements engineering, platform-aware feedback controller design, CPS design specification, CPS design refinement through simulation, service-based decomposition of CPS design, service publication and discovery, resource-aware service deployment, QoS-aware service operation, and service update. Further explanation of these development steps is provided below:

3.1.1 Requirements Engineering

In this development step, requirements of the CPS application and the constraints of the available computing, sensing, and communication platform are documented.

3.1.2 Platform-aware Feedback Controller Design

In traditional feedback control design process, a plant is modeled and a feedback control law is derived using mathematical analysis that assumes either perfect or a worst-case performance of the runtime computing and communication infrastructure. However, in this development step, a feedback control law is developed that provides an active adaptation strategy to respond to various performance levels of underlying communication infrastructure.

3.1.3 CPS Design Specification

In this development step, the result of platform-aware controller design process is captured as a CPS design specification that specifies the physical plant as well as networked controller aspects of a CPS design. Moreover, it also describes the feedback control adaptation strategy to handle the imperfect performance of runtime computing and communication platform. This CPS design specification also serves as an interface between the control engineer and computer systems engineer.

3.1.4 CPS Design Refinement through Simulation

In traditional feedback control design, an initial feedback control law, developed using mathematical analysis, is refined through a simulation environment such as Simulink [51]. Similarly, in this development step, a cyber-physical co-simulation environment is used to refine a CPS design by simulating the performance of the proposed CPS design under various realistic operating conditions of the runtime computing and communication platform. Parameters of the proposed CPS design are tweaked until it shows satisfactory performance for the realistic operating conditions of the runtime computing and communication platform in the cyber-physical co-simulation environment.

3.1.5 Service-based Decomposition of CPS Design

In this development step, a set of service descriptions are generated from the CPS design that was specified earlier in the development process. A service description specifies the following: (1) messages that a service exchanges with other services, (2) sensing and control actions that a service takes on the co-located physical entities, (3) quality-of-service constraints (QoS) constraints on message exchanges with other services, (4) platform resource requirements of a service, and (5) various modes of operation of a service for various QoS fault scenarios.

3.1.6 Service Publication and Discovery

In this development step, service descriptions are published to one or more service repositories. These services are then discovered by appropriate computing nodes. This process of service publication and discovery could be performed offline or online depending on the nature of CPS application.

3.1.7 Resource-aware Service Deployment

In this development step, a service-based computing platform is ported to all the heterogeneous computing nodes involved in the CPS scenario. Then, each computing node accesses its service repository to access its associated service descriptions, which are then deployed on the computing node in a resource-aware manner. If the computing node does not have sufficient resources, service deployment fails. This ensures that any resource constraints in the system are captured at the deployment time and there are no surprise timing failures of CPS application at run time due to resource constraints.

3.1.8 QoS-aware Service Operation

During the service operation, services interact with co-located physical entities through sensing and control actions. Services also interact with each other by sending messages to each other. Moreover, during this step, services switch between different modes of operation if QoS violations occur during message exchange.

3.1.9 Service Update

If the CPS application needs to be updated at some point during its life cycle, a service update step could be carried out. In this step, services again pass through service publication, discovery, resource-aware service deployment, and QoS-aware service operation phases.

3.2 Formal Performance Guarantees

The proposed CPS reference model requires the existence of formal guarantees for the following aspects:

1. functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment.

2. non-interference between the co-deployed services from the perspective of their timing performance.

The above mentioned formal guarantees enable a provably-correct process of converting a CPS application from a CPS design specification to a service-based CPS deployment in the field without affecting the performance of already deployed CPS applications on the same CPS computing platform. Hence, CPS development effort can focus on the platform-aware feedback controller design and simulation-based design refinement. Once the performance of a CPS design has been found to be satisfactory in these two steps, the CPS design can be transformed into a service-based

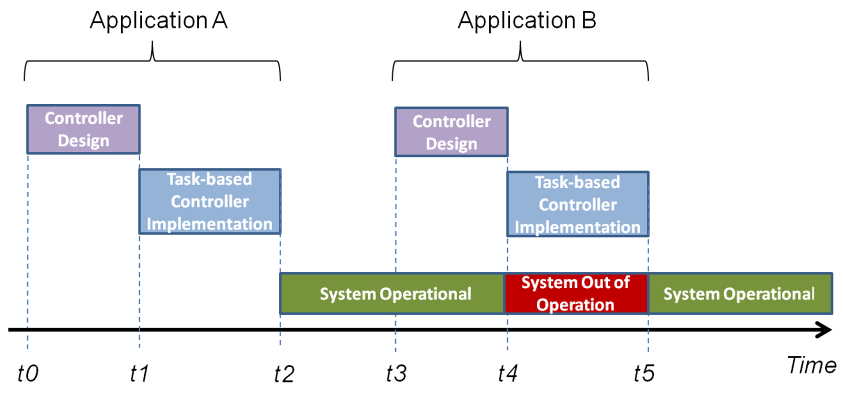


Figure 3.2: Incremental deployment of applications: task-based reference model for embedded control systems (Notice the disruption in system operation during period $[t4, t5]$).

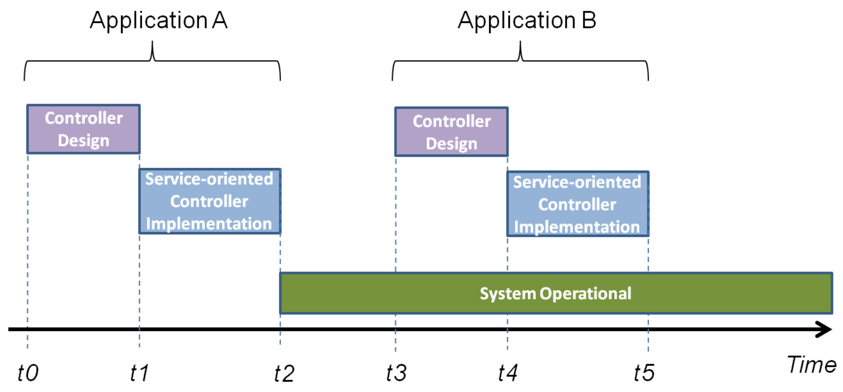


Figure 3.3: Incremental deployment of applications: service-oriented CPS reference model.

field deployment in an automated and provably-correct manner, without worrying about its effects on the existing applications supported by the same CPS computing platform. Therefore, inherent availability of above mentioned formal guarantees in the proposed CPS reference model will enable continuous system evolution, reconfiguration, and maintenance for safety-critical but large scale and "always-online" CPS application domains such as smart grid.

3.3 Advantages over Task-based Reference Model

Through the development steps and inherent formal guarantees outlined above, the proposed service-oriented CPS reference model can address most of the challenges being faced by traditional task-based embedded control system development techniques (from the automotive and avionics domain) in the development of emerging wide-area cyber-physical systems such as smart grid [75] and vehicular networks [54]. For instance, as illustrated in Figure 3.2 and Figure 3.3, the proposed service-oriented CPS reference model can support system reconfiguration and update without taking the system out of operation. This is a critical requirement of the emerging wide-area CPS applications such as smart grid, because (unlike automotive and avionics domain) these systems cannot be taken out of operation for the sake of introducing new functionality in the system.

Unlike the task-based reference model, the inherent formal guarantees of the proposed service-oriented CPS reference model also ensure that in case of an update to the system, the system does not need to be tested from scratch (at the time of system upgrade) as any new service deployments are formally guaranteed to not affect the performance of already deployed services.

CHAPTER IV

TECHNOLOGICAL REQUIREMENTS OF SERVICE-ORIENTED REFERENCE MODEL FOR CYBER-PHYSICAL SYSTEMS

As noted earlier in this dissertation, the reference model for a domain enables the development of a consistent set of technologies and tools for that domain [58]. This chapter identifies some technological requirements based on the service-oriented CPS reference model, described in Chapter 3. Later in this dissertation (Chapter 6, Chapter 7, Chapter 8, and Chapter 9), solutions will be presented for the technological requirements identified in this chapter.

Following are some of the major technological requirements based on the proposed service-oriented reference model for cyber-physical systems:

- CPS design specification language.
- simulation environment for CPS design refinement.
- service description language.
- service-based computing platform for CPS computing nodes with support for resource-aware service deployment and QoS-aware service interaction.
- automated model transformation tool that generates a set of functionally equivalent service descriptions from a CPS design description.

It must be emphasized that the concept of a reference model and associated technological requirements allows the research community to investigate and compare multiple solution approaches for meeting these technological requirements [58].

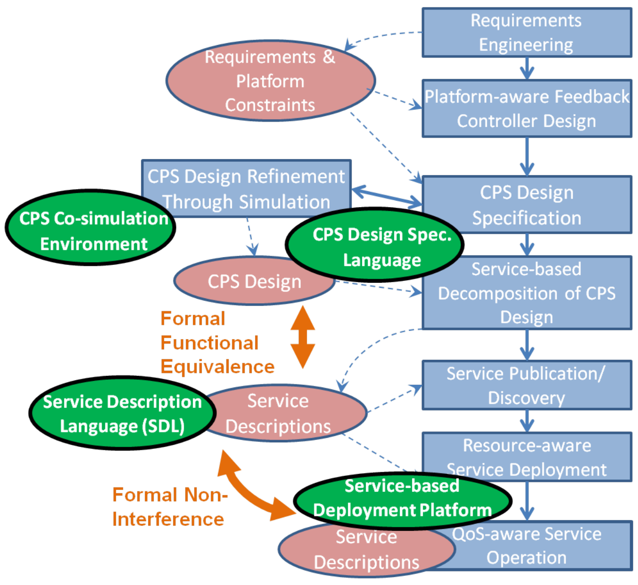

Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Co-simulation Environment
CPS Design Spec. Language
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Description Language (SDL)
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
Service-based Deployment Platform
Service Descriptions
QoS-aware Service Operation

Figure 4.1: Technological requirements of a service-oriented reference model for cyber-physical systems.

Therefore, there could be multiple candidate solutions for meeting each of the technological requirements of the proposed CPS reference model, identified in this chapter. However, any set of solutions for the above mentioned technological requirements of the proposed CPS reference model must ensure the existence of formal guarantees for the following aspects: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and (2) non-interference between the co-deployed services from the perspective of their timing performance.

Figure 4.1 shows the role played by the technological requirements, identified in this chapter, during a CPS development process according to the proposed reference model. Further details of these technological requirements are provided below:

4.1 CPS Design Specification Language

According to the proposed CPS reference model, a CPS design specification captures the results of platform-aware feedback controller design process. Moreover, this CPS design specification also serves as input for the processes of design refinement through simulation and decomposition of CPS design into a set of functionally equivalent service descriptions. In order to develop a CPS design specification that can meet the above mentioned requirements, an appropriate CPS design specification language (CPS-DSL) is required.

4.2 Simulation Environment for CPS Design Refinement

According to the proposed CPS reference model, a CPS design, developed through a platform-aware feedback controller design process, must be refined further through simulation. This design refinement step requires the availability of an appropriate cyber-physical co-simulation environment that can load a CPS design specification and show its performance under various realistic operating conditions of the runtime computing and communication infrastructure.

4.3 Service Description Language

According to the proposed CPS reference model, a service description plays a central role. Once a mature CPS design has been developed through the processes of platform-aware feedback controller design and simulation-based design refinement, this CPS design is decomposed into a set of interacting services, each with its own service description. These service descriptions must specify the following information:

4.3.1 Service Interface

The *service interface* section of a service description describes the messages that the service exchanges with other services and sensing and control actions that a service takes on the co-located physical entities. This section also identifies the QoS constraints on these messages and sensing and control actions.

4.3.2 Service Resources

The *service resources* section of a service description describes platform resource requirements of a service in order to satisfy the QoS constraints identified in the *service interface* section.

4.3.3 Service Modes

Unlike traditional embedded control system domains (such as automotive and avionics systems), some emerging CPS application domains (such as smart grid) are wide-area systems. As a result, QoS constraints on message exchange among computing nodes of a CPS scenario in these domains cannot be guaranteed by the communication subsystem. Therefore, service description for a service must contain a section which defines different modes of operation of the service for different QoS-fault scenarios.

In order to develop service descriptions that contain the above mentioned information (*service interface*, *service resources*, and *service modes*), an appropriate service description language (SDL) is required.

4.4 Service-based Computing Platform for CPS Computing Nodes

To enable CPS development according to the proposed reference model, each CPS computing node must have an appropriate service-based computing platform that can support resource-aware service deployment and QoS-aware service operation. Generally, a CPS scenario involves a set of heterogeneous computing nodes with different processors, operating systems, and middleware technologies. Therefore, the required service-based computing platform must be capable of being ported to these heterogeneous computing nodes.

The resource-aware deployment of a service on a computing platform, as suggested by the proposed reference model, requires the existence of an appropriate service compiler as a part of the service-based computing platform. This service compiler must be capable of reading the service description (specified using an appropriate service description language) and deciding whether a certain computing nodes has enough resources to successfully deploy this service such that the service can meet its

QoS constraints.

4.5 Automated Model Transformation between CPS Design Specification and Service Descriptions

According to the proposed CPS reference model, a CPS design, represented by its CPS design specification, is decomposed into a functionally equivalent set of services, each represented by its own service description. The existence of an automated model transformation tool that can translate a CPS design specification into a functionally equivalent set of service descriptions, can be really useful to streamline the process of CPS development.

CHAPTER V

CASE STUDY: SMART GRID

The need for incorporating environmentally sustainable energy sources into the existing energy mix has resulted in a set of worldwide initiatives towards the development of a smart electric grid [80]. These initiatives aim to overlay the existing electric grid with a more extensive sensing, communication, and computation infrastructure that can enable the grid to handle a higher penetration of intermittent, distributed renewable energy resources without compromising the reliability of service. Implementation of the proposed vision for smart grid will result in a wide-area embedded control system with unprecedented complex interactions between the power infrastructure and accompanying cyber infrastructure [75].

Development of reliable smart grid applications through the traditional task-based approach for embedded control systems will result in prohibitively high development and maintenance costs, because the task-based approach is unable to support disruption-free incremental system deployment and reconfiguration that are fundamental requirements for handling the larger scale, longer life-cycle, and "always-online" nature of smart grid. Therefore, smart grid provides an excellent application domain for illustrating the utility of the service-oriented CPS reference model and associated technologies, presented in this research.

This chapter describes a smart grid scenario that will be used as a case study in the subsequent chapters to not only explain the details of various technological elements associated with the proposed service-oriented CPS reference model, but also demonstrate their advantages over the technologies associated with task-based reference model. The smart grid scenario consists of a 24-bus system, shown in Figure 5.1,

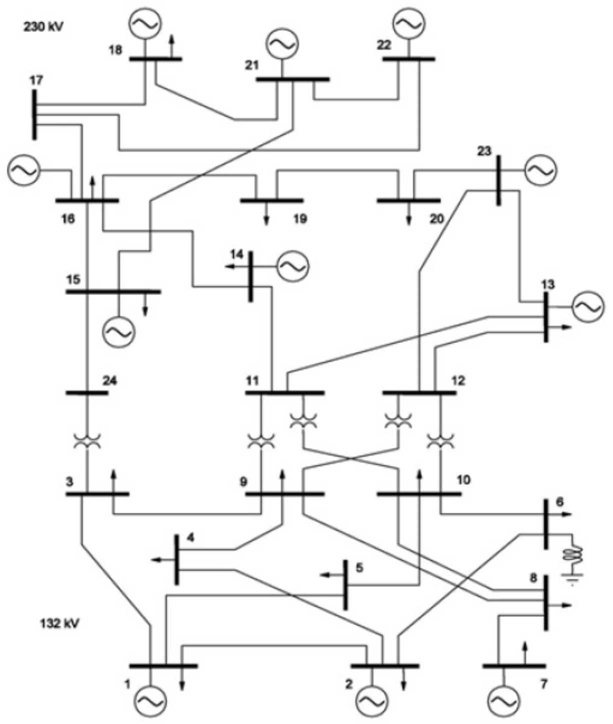


Figure 5.1: IEEE 24-bus case [28].

with two smart grid applications: demand response [1] and power agreement [62]. Demand response application is deployed first, and after a period of successful operation of this application, power agreement application is deployed using the same computing infrastructure. This smart grid scenario has been designed in such a way that it is simple enough to clearly convey the details of the proposed CPS technologies without requiring expertise in the domain of power systems, yet it contains all the elements of a typical wide-are embedded control system that are needed to demonstrate the usefulness of the proposed service-oriented CPS reference model over traditional task-based reference model.

5.1 Demand Response Application

Demand response is a simple but canonical example of a smart grid application. Through demand response application, utilities try to shape elastic load by directly controlling some assets at the consumer premises or by sending price signals to the

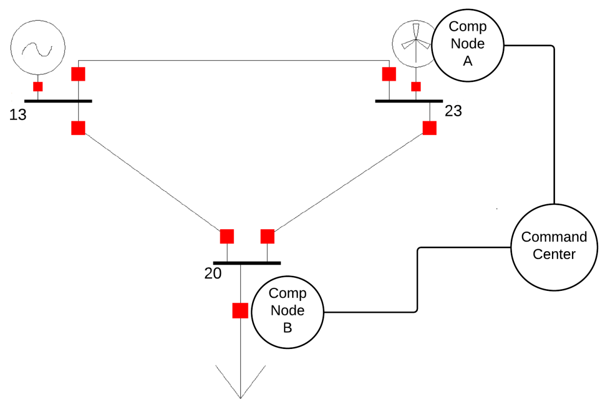


Figure 5.2: Demand response application.

consumer [1].

According to the smart grid scenario under consideration, a direct-control demand response application tries to make the elastic load at Bus_20 follow the ever-changing power output of a wind generator at Bus_23. The wind power profile assigned to the wind generator at Bus_23 is based on the data from a National Renewable Energy Laboratory (NREL) report, which provides mean and standard deviation of one-second wind power step change for a 14-turbine string of 138 turbine wind farm, located in the Buffalo Ridge region of southwest Minnesota [78]. Figure 5.2 shows the relevant power and cyber system topology of the demand response scenario under consideration. The cyber system topology consists of three computing nodes: CompNodeA (co-located with wind generator at Bus_23), CompNodeB (co-located with controllable load at Bus_20), and a CommandCenter.

5.2 Power Agreement Application

Traditionally, electricity grid has been operated by electric utilities using a centralized paradigm in which large-scale generation plants are adjusted from a control center to meet the requirements of ever changing power consumption by the customers. However, due to various renewable energy initiatives, large amounts of customer-end

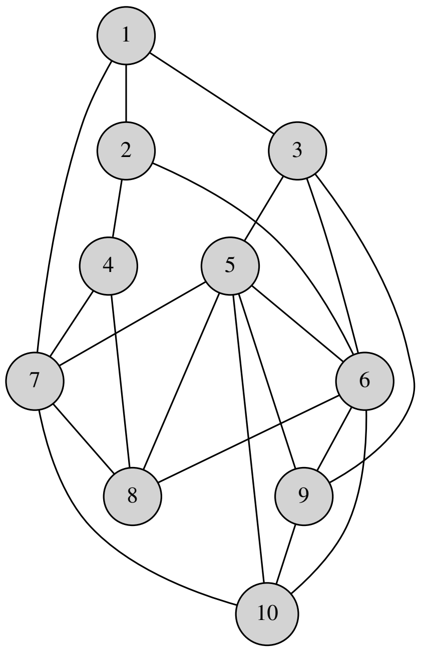


Figure 5.3: Prosumer network graph.

distributed generation and storage resources are expected to be deployed in near future. Application of centralized control paradigm for managing these small-scale distributed energy resources (DER) will result in intractably large control and optimization problems. Due to this limitation of traditional centralized control paradigm, there is growing interest in the distributed control paradigm for power systems [22]. Various research initiatives are underway to develop distributed algorithms for the traditional operating tasks of a power system such as unit commitment [18], economic dispatch [13], and frequency regulation [55].

According to the smart grid scenario under consideration, a recently reported distributed algorithm for smart grid is employed on the 24-bus system in the form of a power agreement application [62]. In the domain of distributed control of smart grid, power system is usually divided into a set of independent control agents. These control agents are also referred to as *prosumers* [22]. In this case study, the 24-bus case diagram is divided into 10 prosumers, as shown in Table 5.1. Figure 5.3 shows the resulting graph; each node in this graph represents a prosumer and each edge

Table 5.1: Division of IEEE 24-bus Case into 10 Prosumers

Prosumer	Buses
1	1,5
2	2,4
3	3,7,24
4	6,13
5	8,16,18
6	9,21
7	10,23
8	11,12,14
9	15,17
10	19,20,22

in the graph shows that the two prosumers (represented by the nodes at the two ends of the edge) are neighbors. In prosumer-based distributed control of smart grid, two prosumers are considered neighbors if there is a branch going from a bus in one prosumer to a bus in the other prosumer.

The distributed power agreement algorithm for a prosumer network, as detailed in [62], considers a set of N prosumers, where each prosumer has computed its desired (or required) power need P_n by taking into consideration its local load, generation, and storage capabilities. As in a physical power network, the power generation and consumption must be balanced, a prosumer cannot consume or produce power in isolation. Therefore, these N prosumers must first co-ordinate (i.e. solve a distributed power agreement problem) to come up with the actual power $\widetilde{P_n}$ that should be produced by each prosumer. In [62], this problem has been formulated as a constrained optimization problem, which minimizes the weighted least squares sum of residuals (between desired power P_n and actual power $\widetilde{P_n}$) subject to a power conservation constraint. Moreover, a decentralized control law to solve this optimization problem has also been presented in [62]. Figure 5.4 summarizes this distributed control law for power agreement in prosumer networks.

For the distributed solution of power agreement control law, presented in [62], participating prosumers go through a series of iterations, consisting of information

$$\boxed{\dot{q} = -LWL\,q + LWP}$$

$$P = Required\ Power = \begin{bmatrix} P_1 \\ P_2 \\ . \\ . \\ P_n \end{bmatrix} \qquad q = Potentials = \begin{bmatrix} q_1 \\ q_2 \\ . \\ . \\ q_n \end{bmatrix}$$

$$L = Graph\ Laplacian\ Matrix \qquad W = Weight\ Matrix = I$$

$$\boxed{\tilde{P} = L\,\tilde{q}}$$

$$\tilde{P} = Agreed\ Power = \begin{bmatrix} \widetilde{P_1} \\ \widetilde{P_2} \\ . \\ . \\ \widetilde{P_n} \end{bmatrix} \qquad \tilde{q} = Final\ Potentials = \begin{bmatrix} \widetilde{q_1} \\ \widetilde{q_2} \\ . \\ . \\ \widetilde{q_n} \end{bmatrix}$$

Figure 5.4: Summary of power agreement control law [62].

exchange and local computations, before converging to the agreed actual power $\widetilde{P_n}$ to be generated by each prosumer. During each iteration, a prosumer needs to know the required power of its 1-hop neighbors and potentials of 2-hop (or less) neighbors. In the prosumer-based distributed operation of power system, power agreement algorithm must run periodically, say every 5 minutes. Therefore, a single run of distributed power agreement algorithm must converge in a reasonably short span of time, say 30 seconds.

Figure 5.5 shows the computing node topology for a corresponding prosumer network. Each prosumer has a ProsumerCompNode that is responsible for local sensing, computation, and control as well as the information exchange with other prosumers in order to successfully implement the distributed power agreement algorithm.

5.3 Incremental Co-deployment of Smart Grid Applications

Although the development of individual smart grid applications (such as demand response and power agreement) is an interesting test case for any CPS development methodology, the application domain of smart grid poses many additional CPS challenges due to its long lifecycle and "always-online" nature. In particular, unlike an

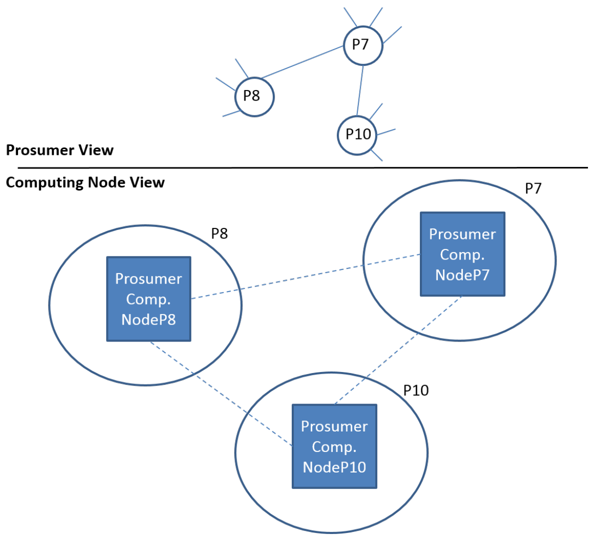


Figure 5.5: Computing nodes for a prosumer network.

automotive or avionic system, an existing smart grid system cannot be taken offline for introducing new functionality [75].

In order to capture these additional challenges of a wide area embedded control system, this smart grid case study assumes that a demand response application has been deployed at a certain time $t0$ and is operating successfully. Then, at a later time instant $t1$, the power agreement application is deployed using the same computing infrastructure. As a result, CompNodeA from Figure 5.2 and ProsumerCompNodeP7 from Figure 5.5 are implemented using the same computing node, while CompNodeB from Figure 5.2 and ProsumerCompNodeP10 from Figure 5.5 are also implemented using the same computing node.

CHAPTER VI

CPS DESIGN SPECIFICATION LANGUAGE

According to the proposed CPS reference model, a CPS design specification captures the results of platform-aware feedback controller design process. Moreover, this CPS design specification also serves as input for the processes of design refinement through simulation and decomposition of CPS design into a set of functionally equivalent service descriptions. In order to develop a CPS design specification that can meet the above mentioned requirements, an appropriate CPS design specification language (CPS-DSL) is required. Figure 6.1 shows the role played by the CPS-DSL in the context of the proposed service-oriented CPS reference model.

6.1 Requirements

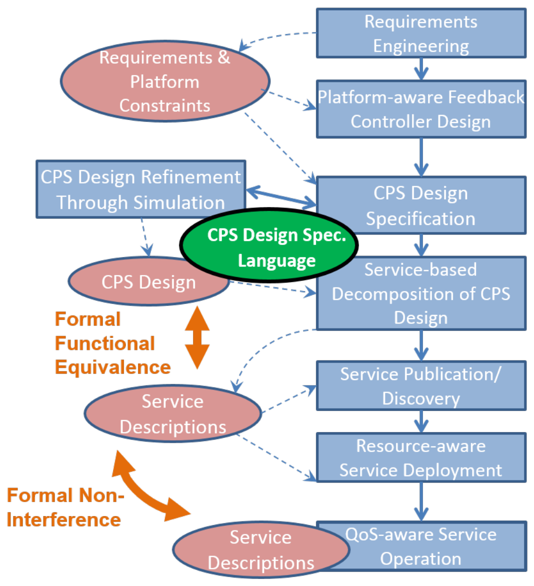
Following are some of the major requirements that a CPS design specification language (CPS-DSL) must meet:

6.1.1 Physical Plant Specification

An appropriate CPS-DSL must have the capability to describes the the physical plant of a CPS through a combination of atomic elements of that physical plant. Moreover, CPS-DSL must clearly identify the physical plant parameters that are sensed or actuated upon by the feedback controller.

6.1.2 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



Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Design Spec. Language
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Descriptions
Service Publication/ Discovery
Resource-aware Service Deployment
Formal Non-Interference
Service Descriptions
QoS-aware Service Operation

Figure 6.1: Role of the CPS Design Specification Language (presented in this chapter) in the service-oriented reference model for cyber-physical systems.

different control nodes.

6.1.3 Specification of Controller Adaptation Strategies

As described earlier, for the emerging wide-area CPS application domains, 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.

6.1.4 Interface between Control Engineer and Real-time Computer Systems Engineer

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

6.2 Design

This section presents various aspects of a proposed CPS-DSL that can meet the requirements identified in Section 6.1 . In particular, various language elements, concrete syntax, abstract syntax, and semantics of the proposed CPS-DSL are described.

6.2.1 Language Elements

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*.

6.2.1.1 Physical System Elements

CompoundPhysicalPlant, *AtomicPhysicalPlant*, *PhysicalSystemParameter* and *PhysicalLink* elements belong to the category of *physical system elements*. Physical plant component of a CPS design can be specified by a set of *AtomicPhysicalPlant* elements connected to each other through *PhysicalLink* elements. A set of *AtomicPhysicalPlant* and *PhysicalLink* elements can also be grouped together into a *CompuondPhysicalPlant* element. Moreover, *PhysicalSystemParameter* elements are used to identify the parameters of a physical plant that are to be sensed and actuated upon by the cyber system.

6.2.1.2 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.

6.2.1.3 Cyber System Elements

ComputingNode, *CommunicationNetwork*, *ControlApp*, *SensorPort*, *ActuatorPort*, *InputMsgPort*, *OutputMsgPort*, *Mode*, *ModeSwitchLogic*, *ControllerFunction*, *PeriodicControllerInput*, and *PeriodicControllerOutput* 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.

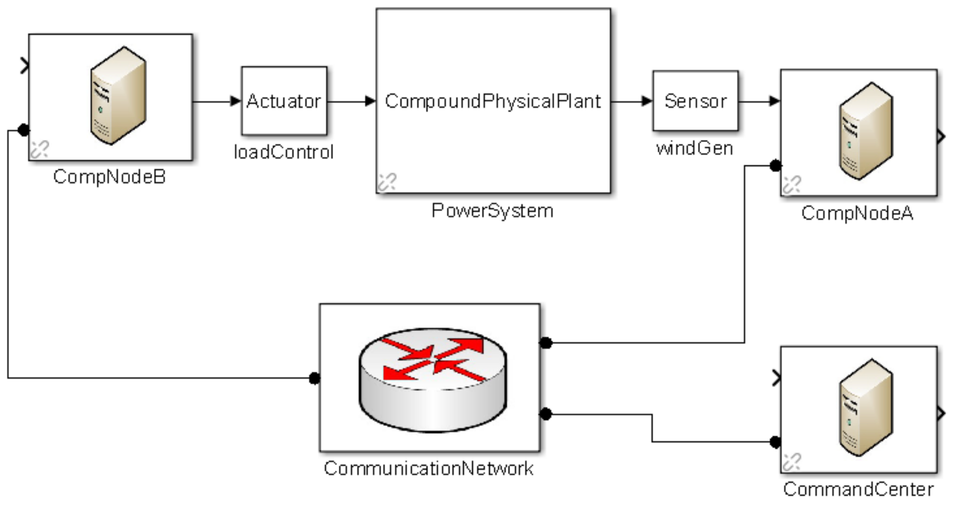


Figure 6.2: A CPS design, specified as Simulink model with the proposed CPS-DSL.

6.2.2 Concrete Syntax

Since Simulink [51] (combined with auxiliary Stateflow [52] and Simscape [29] blocks) has become a defacto 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 [70] has been developed that provides a Simulink block for each element of the proposed CPS-DSL, described in Section 6.2.1. 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 6.2 shows a Simulink model that specifies a CPS design using the Simulink-based concrete syntax of the proposed CPS-DSL. Figure 6.3 shows the internal details of a *ComputingNode* block, which contains a *ControlApp* block and a set of *SensorPort*, *ActuatorPort*, *InputMsgPort*, and *OutputMsgPort* blocks. Figure 6.4 shows the internal details of *ControlApp* block, which consists of a set of *Mode* blocks and a *ModeSwitchLogic* block. Figure 6.5 shows the internal details of *Mode* block,

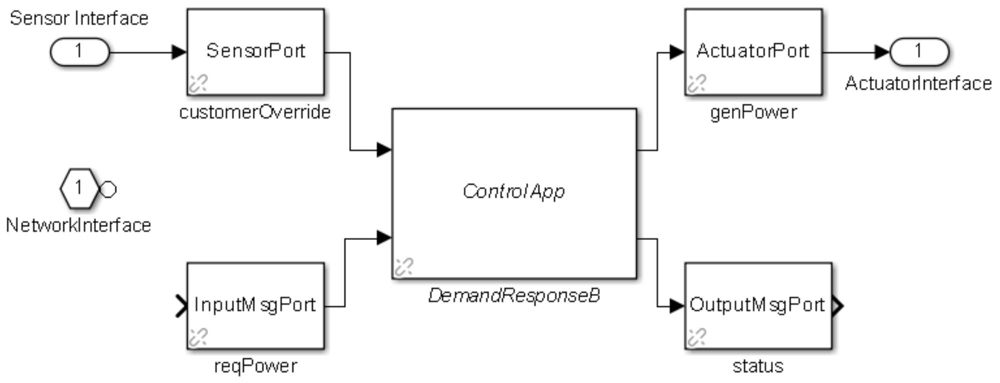


Figure 6.3: Internal details of *ComputingNode* block, named CompNodeB, in Figure 6.2.

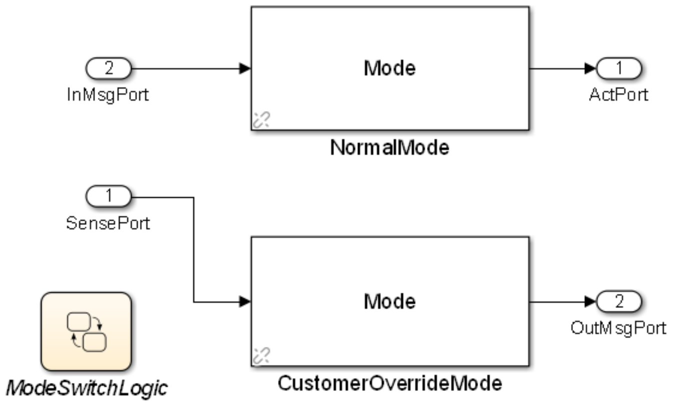


Figure 6.4: Internal details of *ControlApp* block, named DemandResponseB, in Figure 6.3.

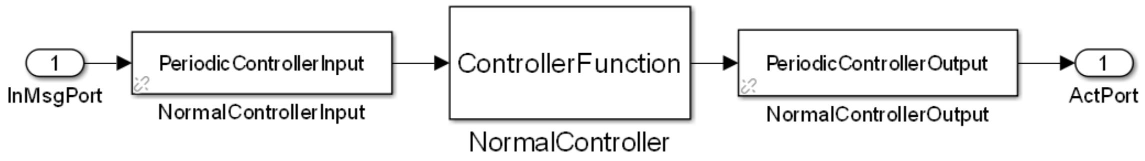


Figure 6.5: Internal details of *Mode* block, named NormalMode, in Figure 6.4.

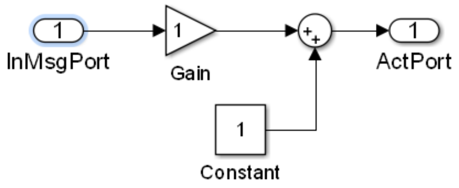


Figure 6.6: Internal details of *ControllerFunction* block, named NormalController-Function, in Figure 6.5.

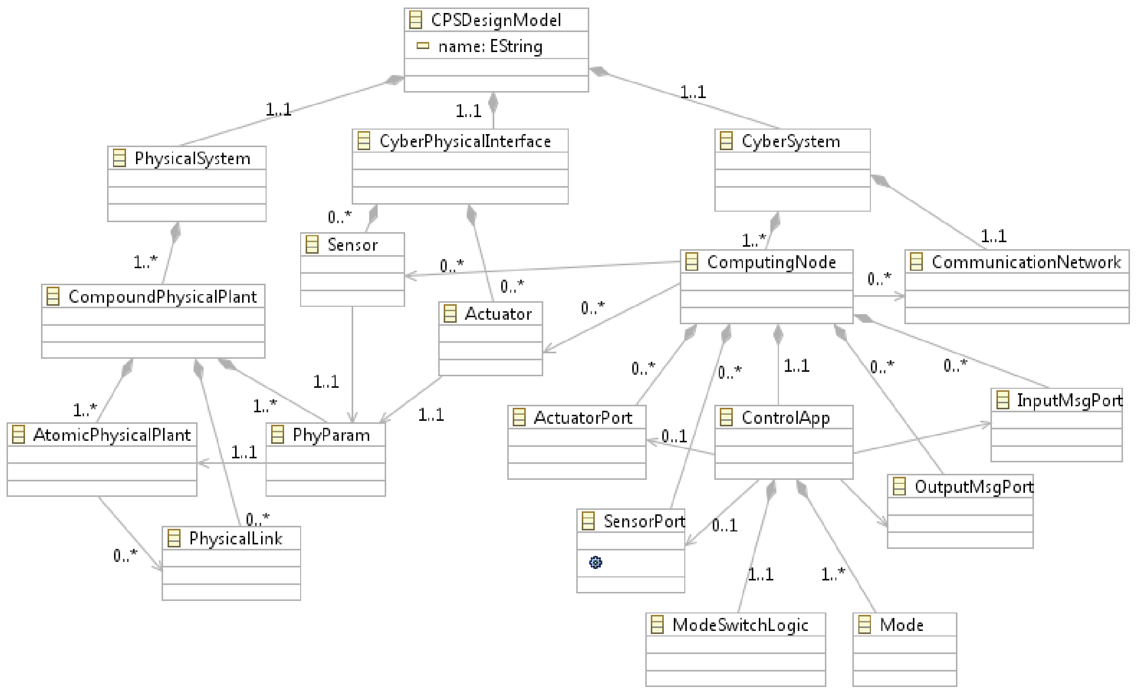


Figure 6.7: Ecore-based meta-model of proposed CPS-DSL.

which contains a set of *ControllerFunction*, *PeriodicControllerInput*, and *Periodic-ControllerOutput* blocks. Figure 6.6 shows the internal details of *ControllerFuncton* block, which contains a description of feedback control law using standard Simulink computation blocks.

6.2.3 Abstract Syntax

Abstract syntax of the proposed CPS-DSL has been implemented as an Ecore-based meta-model [23]. Ecore meta-modeling language was originally developed as a part of Eclipse Modeling Framework (EMF) project [67]. Figure 6.7 shows a the simplified version of the Ecore-based meta-model for the proposed CPS-DSL.

6.2.4 Semantics

According to the semantics of the proposed CPS-DSL, at a given time, only one *Mode* element inside a *ControlApp* is active. 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. Moreover, a *ControllerFunction* element takes time Δt to transfer any change in its input to its output where $0 < \Delta t < T$.

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 and the language used to describe the behavior of an *AtomicPhysicalPlant* element. This capability makes the proposed CPS-DSL more flexible. However, for the rest of this dissertation, it will be assumed that Simulink computation blocks are used to define the control law computation inside a *ControllerFunction* element and Simulink physical system modeling blocks are used to describe the behavior of an *AtomicPhysicalPlant* element.

6.3 Case Study

This section shows the application of the proposed CPS-DSL for design specification of CPS applications, involved in the smart grid case study, presented in Chapter 5. Figure 6.2 shows the top-level diagram for a Simulink model that specifies the design of demand response application, discussed in Section 5.1. Moreover, Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6 show the internal details of the Simulink model, describing the design of demand response application. Design of the power agreement application, discussed in Section 5.2, can also be specified by developing a similar Simulink model to the one depicted in Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6.

CHAPTER VII

SIMULATION ENVIRONMENT FOR CPS DESIGN REFINEMENT

According to the proposed CPS reference model, a CPS design, developed through a platform-aware feedback controller design process, must be refined further through simulation. This design refinement requires the availability of an appropriate cyber-physical co-simulation environment that can load a CPS design specification and show its performance under various conditions of the runtime communication infrastructure. Figure 7.1 shows the role played by a simulation environment for CPS design refinement in the context of the proposed service-oriented CPS reference model.

7.1 Requirements

Following are the major required characteristics of an appropriate simulation environment that can be used for simulation-based CPS design refinement process.

7.1.1 Co-simulation of Physical and Cyber Subsystems

An appropriate simulation environment for CPS design refinement must be capable of simulating both the cyber and physical aspects of the system. Moreover, such a simulation environment must also faithfully capture the interaction between cyber and physical components of the system that results from the sensing and actuation process involved in a cyber-physical system.

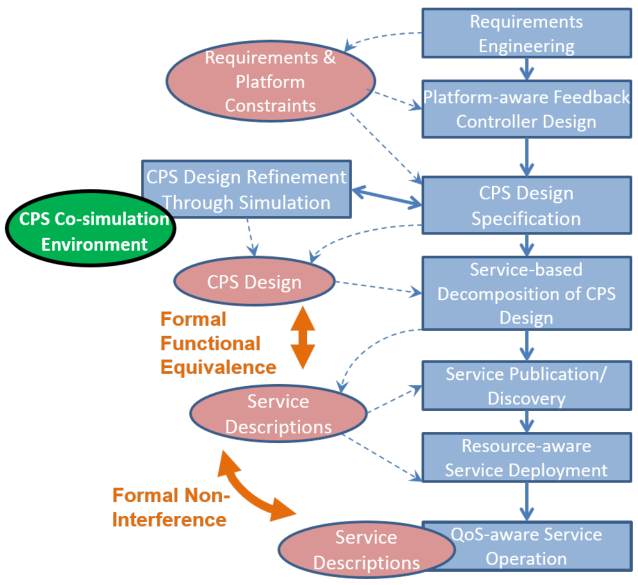

Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Co-simulation Environment
CPS Design Specification
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
Service Descriptions
QoS-aware Service Operation

Figure 7.1: Role of the simulation environment for CPS design refinement (presented in this chapter) in the service-oriented CPS reference model.

7.1.2 Simulation of Computer Networks

Since the cyber subsystem of a CPS consists of a set of networked computing nodes, a suitable simulation infrastructure for CPS design refinement must simulate the computer network involved in the CPS. Various popular network simulators are available that can be leveraged while developing an appropriate simulation environment for CPS design refinement. Some examples of such network simulators are ns-2 [30], ns-3 [57], and OMNet++ [77].

7.1.3 Simulation of Application-level Software

For simulating the cyber aspects of a CPS, simulation of physical communication layer and networking protocols of a computer network is not sufficient. An appropriate simulation environment for CPS design refinement must also be able to simulate the effects of multi-mode feedback control law, executing as application-level software at different computing nodes of the computer network.

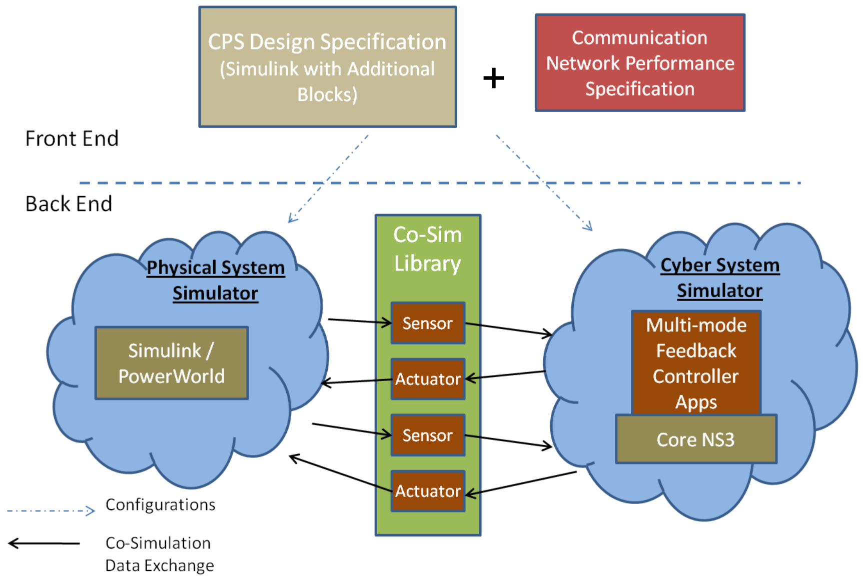
7.1.4 Automated Configuration of Simulation Engine

In the simulation-based design refinement step of traditional development methodology for embedded control systems, designers make extensive use of user-friendly, graphical simulation environments such as Simulink [51]. Therefore, an appropriate simulation environment for CPS design refinement must also provide a similar level of user friendliness by supporting the automated configuration of the underlying simulation engine from an appropriate front-end user interface.

7.2 Design

This section presents the design of an ns-3 based simulation environment for CPS design refinement that has been developed in order to meet the requirements identified in Section 7.1. Figure 7.2 shows the overall structure of this simulation environment for CPS design refinement. The proposed simulation environment extends a state-of-the-art network simulator, ns-3, with a cyber-physical co-simulation library and the support for simulating multi-mode feedback controller applications [57]. The co-simulation library provides a generic interface API, based on the concepts of *sensor* and *actuator*, that has been used to integrate two physical system simulators (Simulink [51] and PowerWorld [60]) with ns-3. Figure 7.3 shows the overall organization of ns-3 software after the additions that have been made to ns-3 as a part of the proposed simulation environment. This simulation environment also includes a Simulink-based front-end that allows the user to provide a CPS design specification (using the CPS-DSL presented in Chapter 6) under consideration and the various communication network scenarios under which the performance of this CPS design must be simulated for the sake of design refinement.

Figure 7.4 shows the UML class diagram that depicts the relationship between the major classes involved in the design of proposed simulation environment. This simulation environment has been implemented by adding three modules to the ns-3



CPS Design Specification (Simulink with Additional Blocks)
+
Communication Network Performance Specification
Front End
Back End
Co-Sim Library
Physical System Simulator
Simulink / PowerWorld
Sensor
Actuator
Sensor
Actuator
Cyber System Simulator
Multi-mode Feedback Controller Apps
Core NS3
Configurations
Co-Simulation Data Exchange

Figure 7.2: Structure of the proposed simulation environment for CPS design refinement.

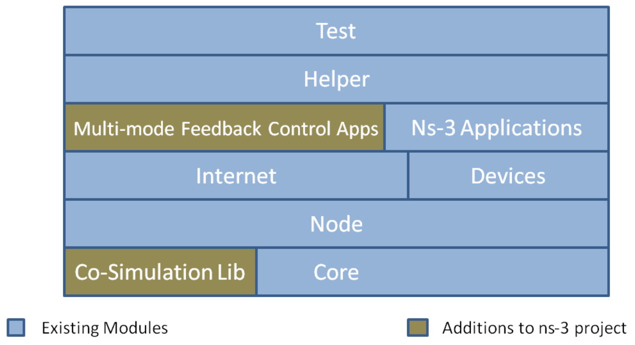


Figure 7.3: Additions to the standard structure of ns-3 network simulator.

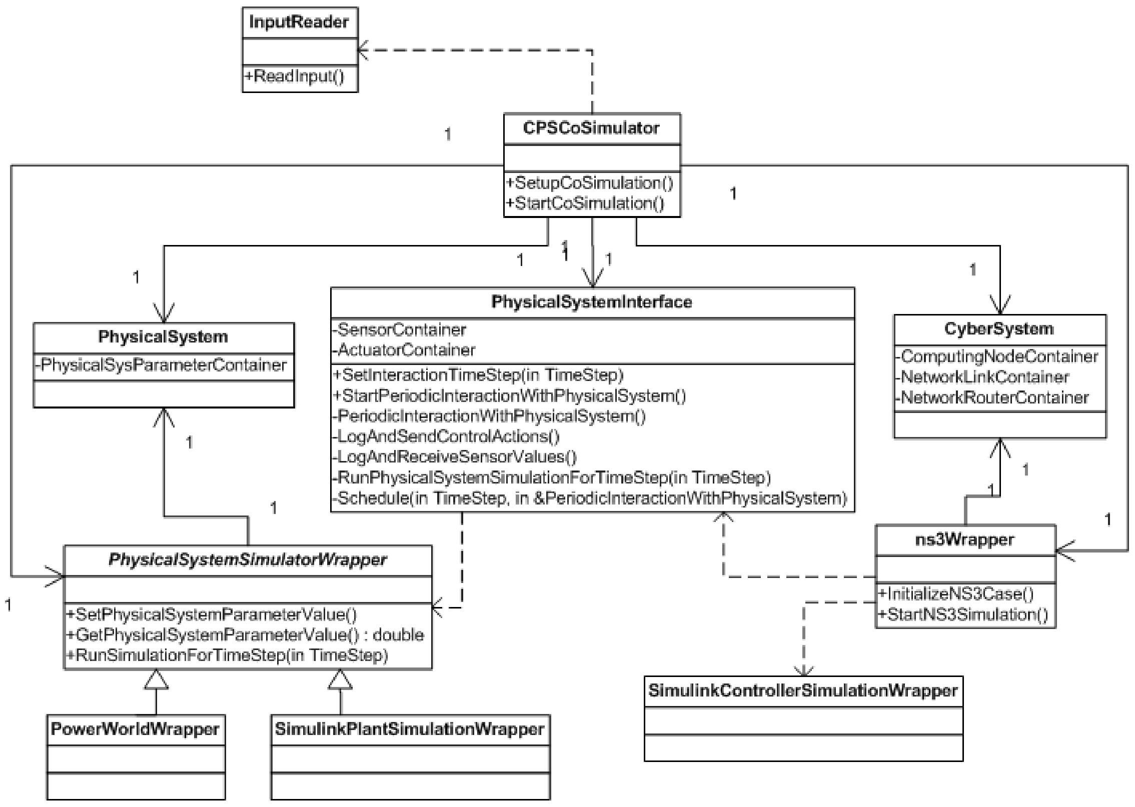

InputReader
+ReadInput()
CPSCoSimulator
+SetupCoSimulation()
+StartCoSimulation()
PhysicalSystem
-PhysicalSysParameterContainer
PhysicalSystemInterface
-SensorContainer
-ActuatorContainer
+SetInteractionTimeStep(in TimeStep)
+StartPeriodicInteractionWithPhysicalSystem()
-PeriodicInteractionWithPhysicalSystem()
-LogAndSendControlActions()
-LogAndReceiveSensorValues()
-RunPhysicalSystemSimulationForTimeStep(in TimeStep)
-Schedule(in TimeStep, in &PeriodicInteractionWithPhysicalSystem)
CyberSystem
-ComputingNodeContainer
-NetworkLinkContainer
-NetworkRouterContainer
PhysicalSystemSimulatorWrapper
+SetPhysicalSystemParameterValue()
+GetPhysicalSystemParameterValue() : double
+RunSimulationForTimeStep(in TimeStep)
ns3Wrapper
+InitializeNS3Case()
+StartNS3Simulation()
PowerWorldWrapper
SimulinkPlantSimulationWrapper
SimulinkControllerSimulationWrapper
1

Figure 7.4: UML class diagram for proposed simulation environment.

code base: a physical system module, a cyber-system module, and a physical system interface module. Physical system module provides two major classes: *PhysicalSystem* and *PhysicalSystemSimulatorWrapper*. The *PhysicalSyste* class contains a list of physical system parameters that are sensed or actuated upon in a CPS scenario and therefore, need to be exchanged between the cyber and physical components of a CPS co-simulator. The *PhysicalSystemSimulatorWrapper* class servers as a generic wrapper around the various physical system simulation environments such as Simulink [51] and PowerWorld [60].

The cyber system module provides two major classes: *CyberSystem* and *NS3Wrapper*. The *CyberSystem* class contains a list of cyber system entities such as computing nodes, network links, and network routers. The *NS3Wrapper* class serves as a wrapper around the standard ns-3 code. The *NS3Wrapper* class sets up the ns-3 simulation scenario based on the information that it receives through a *CyberSystem* object.

Table 7.1: Co-Simulation Procedure

```
void PeriodicInteractionWithPhysicalSystem( )
{
  LogAndSendControlActions( );

  LogAndReceiveSensorValues( );

  RunPhysicalSystemSimulationForTimeStep(TimeStep);

  Schedule(TimeStep, &PeriodicInteractionWithPhysicalSystem);
}
```

The physical system interface module provides four major classes: *Sensor*, *Actuator*, *PhysicalSystemParameter*, and *PhysicalSystemInterface*. In the proposed co-simulation infrastructure, feedback controller algorithms running at the ns-3 application layer use these *Sensor* and *Actuator* objects to interact with the physical system simulator. *Sensor* and *Actuator* classes hold an instance of the *PhysicalSystemParameter* class, which represents the physical system entity sensed by a sensor or actuated upon by the actuator. In this co-simulation infrastructure, an attribute can only be transferred between the two component simulators (ns-3 and physical system simulator) if it is modeled as an instance of *PhysicalSystemParameter* class. In the proposed CPS co-simulator, the interaction between the cyber and physical system simulators is done on a periodic basis through the *PhysicalSystemInterface* class. Table 7.1 shows the method of *PhysicalSystemInterface* class that achieves the periodic interaction between cyber and physical system simulation components.

7.3 Case Study

This section shows the application of the proposed simulation environment for design refinement of CPS applications, involved in the smart grid case study, presented in Chapter 5.

For the analysis of demand response application through the proposed simulation environment, a communication network scenario consisting of star topology of three

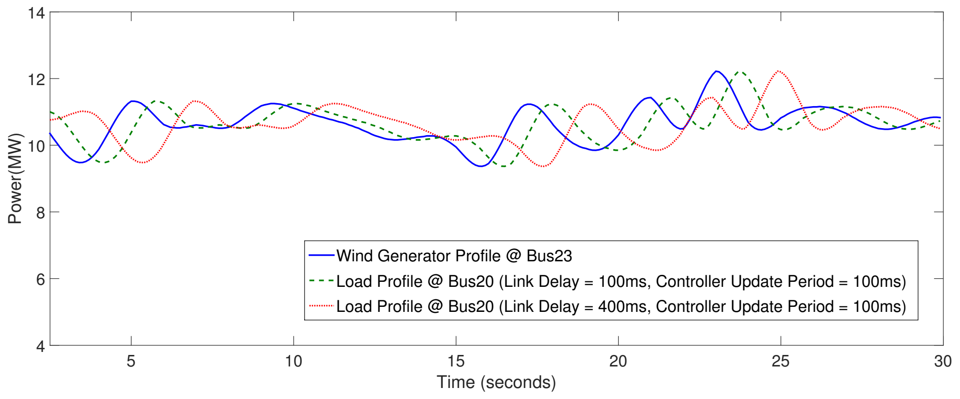


Figure 7.5: Demand response application: power generation and consumption profiles for different link delays.

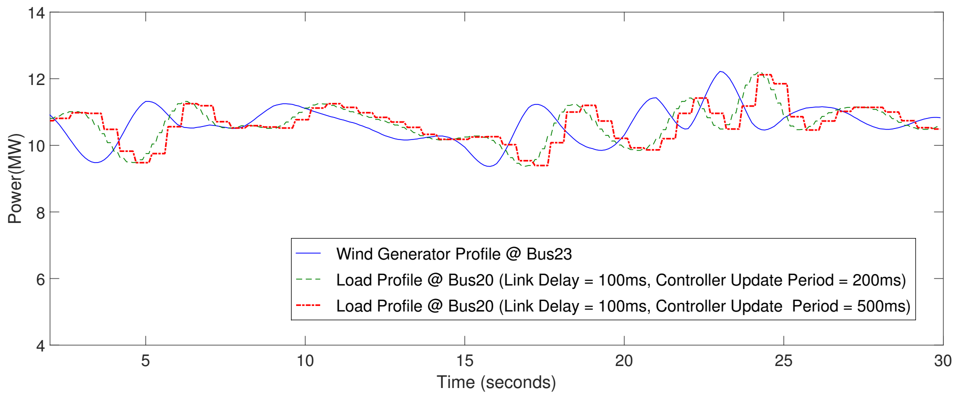


Figure 7.6: Demand response application: power generation and consumption profiles for different controller update periods.

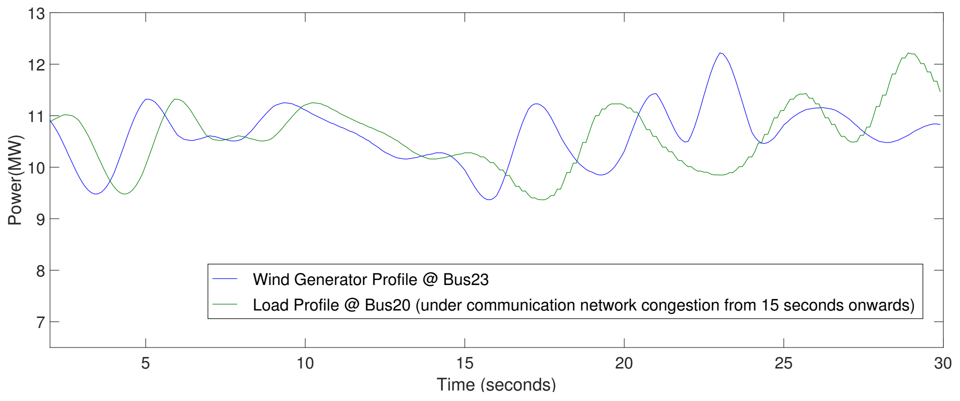


Figure 7.7: Demand response application: power generation and consumption profiles under communication network congestion.

computing nodes (CompNodeA, CompNodeB, and CommandCenter) and a central network router was specified. Furthermore, in this communication network scenario, UDP sockets were used for communication between computing nodes. Figure 7.5 and Figure 7.6 show the performance of demand response application for different communication link delays and different controller update periods respectively, under this communication network scenario. Figure 7.7 shows the simulated performance of demand response application under network router congestion, caused by the addition of an external traffic source in the earlier communication network scenario. These examples illustrate that through the simulation environment presented in this chapter, the performance of a proposed demand response application design can be evaluated under complex (but relevant) cyber system conditions and the design parameters (such as controller update period, controller modes of operation, and mode transition conditions) can be tweaked until satisfactory performance is seen in the simulation.

For the analysis of power agreement application through the proposed simulation environment, a communication network scenario consisting of point-to-point communication links between all the ProsumerCompNodes was specified. Furthermore,

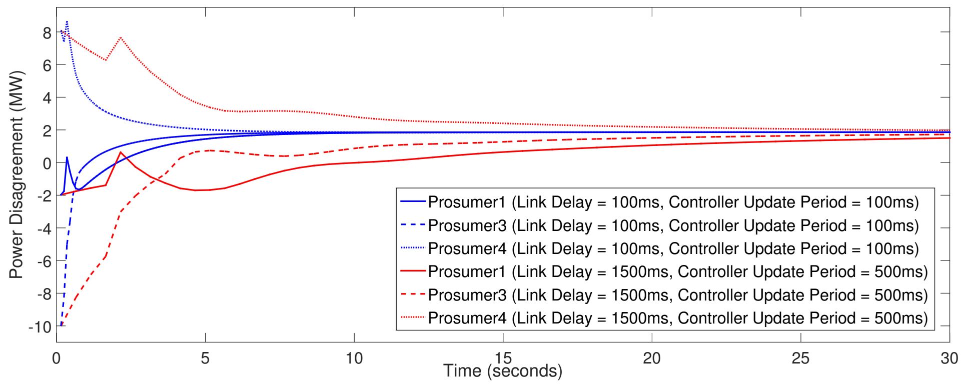


Figure 7.8: Power agreement application: convergence behavior under two different combinations of communication link delay and controller update period (only 3 out of 10 prosumers are depicted for readability).

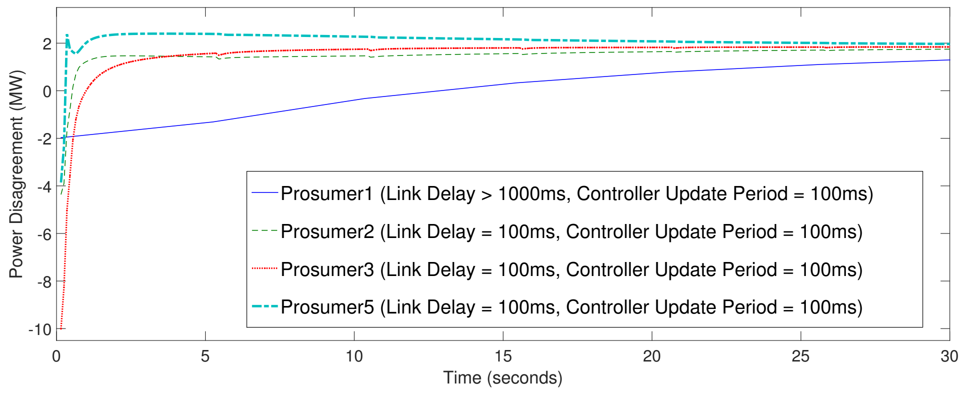


Figure 7.9: Power agreement application: convergence behavior under network congestion on point-to-point communication links to ProsumerCompNode for prousmer1 (only 4 out of 10 prosumers are depicted for readability).

in this communication network scenario, UDP sockets were used for communication between all the ProsumerCompNodes. Figure 7.8 shows the convergence behavior of power agreement application for two different combinations of link delay and controller update period, under this communication network scenario. Firgure 7.9 shows the convergence behavior when point-to-point communication links of ProsumerCompNode for Prosumer1 have significantly less capacity as compared to other point-to-point communication links in the system. This kind of analysis, through the proposed simulation environment, can allow us to investigate whether power agreement application will converge in the required time span of 30 seconds under some complex (but relevant) cyber system scenarios. This information could be useful for refining the different modes of operation and mode transition conditions, defined for the power agreement application during the platform-aware feedback controller design stage.

CHAPTER VIII

SERVICE DESCRIPTION LANGUAGE

According to the proposed CPS reference model, a service description plays a central role. Once a mature CPS design has been developed through the processes of platform-aware feedback controller design and simulation-based design refinement, this CPS design is decomposed into a set of interacting services, each with its own service description. In order to develop these service descriptions, an appropriate service description language (SDL) is required. Figure 8.1 shows the role played by a CPS service description language in the context of the proposed service-oriented CPS reference model.

8.1 Requirements

Any proposed service description language (SDL) must be capable of specifying the following information about a service.

8.1.1 Service Interface

The *service interface* section of a service description describes the messages that the service exchanges with other services and sensing and control actions that a service takes on the co-located physical entities. This section also identifies the QoS constraints on these messages and sensing and control actions.

8.1.2 Service Resources

The *service resources* section of a service description describes platform resource requirements of a service in order to satisfy the QoS constraints identified in the *service interface* section.

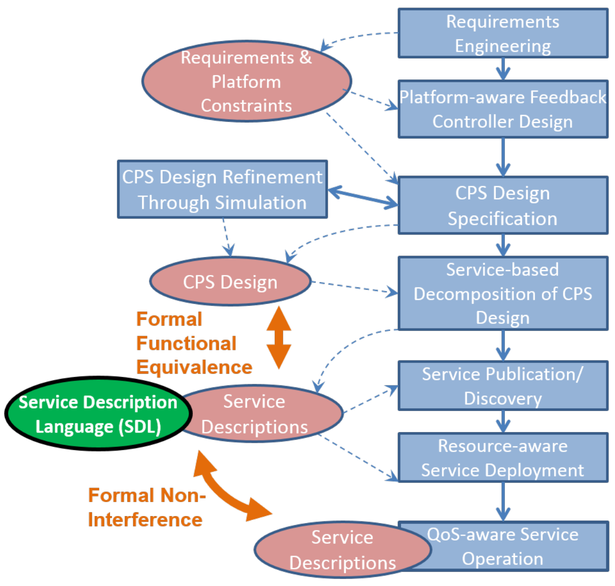

Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Description Language (SDL)
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
Service Descriptions
QoS-aware Service Operation

Figure 8.1: Role of the CPS Service Description Language (presented in this chapter) in the service-oriented CPS reference model.

8.1.3 Service Modes

Unlike traditional embedded control system domains (such as automotive and avionics systems), some emerging CPS application domains (such as smart grid) are wide-area systems. As a result, QoS constraints on message exchange among computing nodes of a CPS scenario in these application domains cannot be guaranteed by the communication subsystem. Therefore, service description for a service must contain a section which defines different modes of operation of the service for different QoS-fault scenarios.

8.2 Design

This section presents the design of a service description language (SDL) for CPS that is capable of specifying all the elements of a service, as outlined in Section 8.1. The syntax and semantics of the proposed SDL are heavily influenced by Giotto language, which was originally proposed as a programming language for embedded control systems [25].

8.2.1 Giotto Programming Language

The typical development process for an embedded control system can be divided into two steps: *control design* and *software implementation.* During the *control design* phase, a control engineer models the plant behavior and disturbances, derives the feedback control laws, and validates the performance of plant under the influence of feedback controller through mathematical analysis and simulations. During the *software implementation* phase, a software engineer breaks down the feedback controller's computational activities into tasks and associated timing constraints on the completion of these tasks. Then, the software engineer develops code for these tasks in a traditional programming language (such as C) and assigns priorities to these tasks so that the tasks could meet their timing constraints while being scheduled on a processor by the scheduler of a real-time operating system (RTOS).

Giotto programming language aims to bridge the communication gap between control engineer and software engineer by providing an intermediate level of abstraction between control design and software implementation [25]. Giotto language syntax can be used by a Giotto program to specify time-triggered sensor readings, actuator updates, task invocations, and mode transitions. Then, a Giotto compiler must be used to compile (an entirely platform independent) Giotto program onto a specific computing platform. The compiler must preserve the functionality as well as the timing behavior specified by the Giotto program. The Giotto compilation process is aided by the use of E Machine [26], a virtual machine that serves as the target for compilation of Giotto programs. Figure 8.2 shows the Giotto and E Machine configuration for a typical networked embedded control system.

Figure 8.3 shows the major elements of Giotto syntax: *task*, *mode*, *driver*, *port*, and *guard*. *Task* is the basic functional unit of Giotto language and represents a periodically executable piece of code. Giotto *tasks* communicate with each other as well as with sensors and actuators. However, in Giotto, all data communication occurs

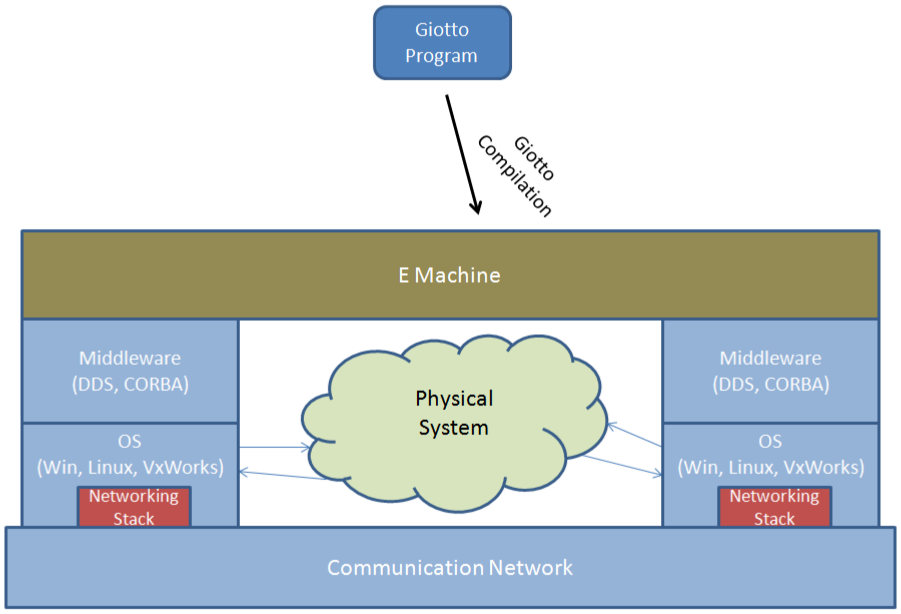


Figure 8.2: Typical configuration of Giotto and E Machine for embedded control systems.

through *ports*. In a Giotto program, there are mutually disjoint sets of task ports, sensor ports, and actuator ports. Task ports are further divided into task input ports, task output ports, and task private ports. Each *task* also has an associated function f (implemented in any sequential programming language) from its input ports and private ports to its output ports and private ports. According to Giotto semantics, sensor ports are updated by the environment while task ports and actuator ports are updated by the Giotto program.

Driver represents a piece of code that transports values between two *ports*. A *driver* can also have an associated *guard*, which is some boolean-valued function on the current values of certain *ports*. The code associated with the *driver* only executes if the *guard* of the *driver* evaluates to *true*. According to Giotto semantics, a *task* is an application-level code that consumes non-negligible amount of CPU time, while *driver* is a system-level code that can be executed instantaneously before the environment changes its state.

At the highest level of abstraction, a Giotto program is essentially a set of *modes*.

At a certain instant of time, Giotto program can only be in one of its *modes*. However, during its execution, a Giotto program transitions from one *mode* to another based on the values of different *ports*. These possible *mode* transitions are specified in Giotto syntax through *mode swithces*. A *mode switch* specifies a target *mode*, switch frequency, and a guarded *driver*. Formally, a Giotto *mode* is made up of several concurrent *tasks*, a set of *mode switches*, a set of mode *ports*, a set of actuator updates, and a period. Each *task* of a *mode* specifies its frequency of execution per *mode* period. While Giotto program is in a certain *mode*, it repeats the same pattern of *task* executions for each *mode* period.

Figure 8.3 shows a Giotto program with two *modes*, m1 and m2. *Mode* m1 has two *tasks*, t1 and t2, while *mode* m2 has only one *task*, t3. *Mode* m1 has a period of 10ms, while *mode* m2 has a period of 20ms. *Task* t1 has a frequency of 2, while *task* t2 has a frequency of 1. This means that as long as Giotto program is in *mode* m1, *task* t1 executes every 5ms while *task* t2 executes every 10ms. Moreover, in this example, there is a *mode switch* from *mode* m1 to *mode* m2 with a switch frequency of 2. This implies that the *mode switch* condition (provided by the *guard* of *driver* d5) is tested every 5ms.

8.2.2 Extensions to Giotto Programming Language

Current syntax of Giotto, summarized in Section 8.2.1, is capable of describing all the elements of a CPS service, except for the input and output messages of a service and QoS constraints associated with these messages. In order to overcome this deficiency, Giotto syntax has been extended with two new types of *ports*: *input message port* and *output message port*. The *input message port* also has the following additional attribute attached it: *TimeSinceLastUpdate*. This attribute could be used in the guard conditions, present in *mode switches*. As a result, the proposed Giotto-based SDL

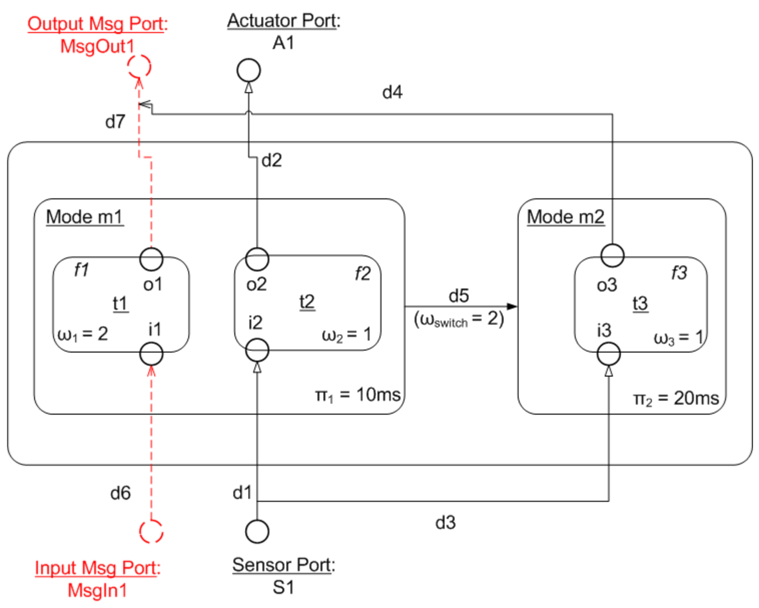


Figure 8.3: Major programming elements of Giotto language. Proposed extensions for a Giotto-based CPS service description language are shown in red with dotted lines.

can be used to specify mode switches based on the violation of QoS constraints associated with message exchanges among services. Figure 8.3 also shows these proposed extensions that result in a Giotto-based CPS service description language.

8.3 Case Study

This section shows the application of the proposed Giotto-based service description language (SDL) for describing CPS services, involved in the smart grid case study, presented in Chapter 5.

Demand response application, involved in the smart grid case study of Chapter 5, can be decomposed into three services: *DemandResponseServiceA*, *DemandResponseServiceB*, and *DemandResponseServiceCC*. Table 8.1 shows the service description of *DemandResponseServiceB* using the proposed Giotto-based service description language, while Figure 8.4 shows the same service description graphically.

DemandResponseServiceB consists of two *modes*: m1 (representing the normal operating mode) and m2 (representing the operating mode when the customer overrides

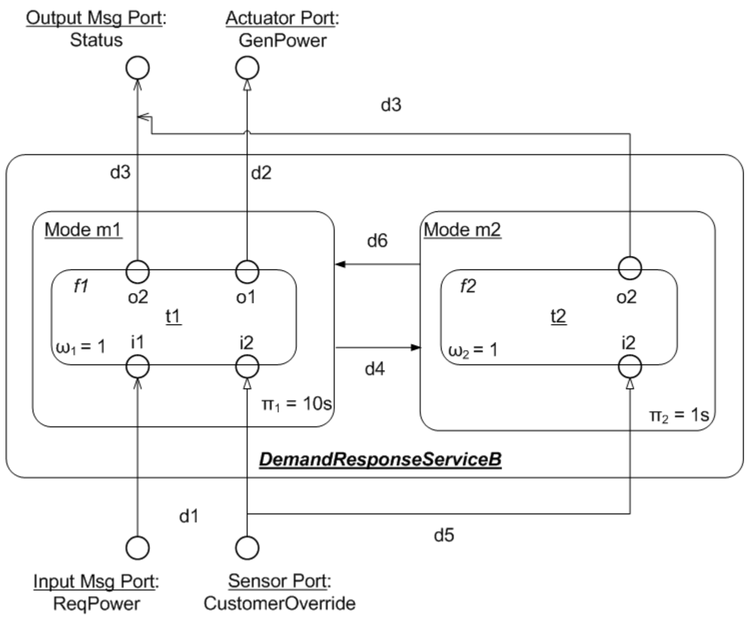


Figure 8.4: Graphical representation of service descriptions for *DemandResponseServiceB.*

the operation of demand response application). *Driver* d4 and *guard* g4 combine to describe the mode switch condition from m1 to m2, while *driver* d6 and *guard* g6 describe the mode switch condition from m2 to m1. Mode transitions between m1 and m2 occur based on the value of sensor port *customerOverride*, which represents the binary status of an application override user interface mechanism available to the customer. According to the service description, shown in Table 8.1, *mode* m1 has a period of 10000ms and it has a *mode switch* with the target *mode* of m2 and a frequency of 1, indicating that the mode switch condition is tested once every mode period. Therefore, mode switch condition from m1 (normal mode) to m2 (user override mode) is tested every 10 seconds.

Table 8.1: Service Description of *DemandResponseServiceB* using the Proposed Giotto-based Service Description Language

```
Sensor Ports
    port customerOverride type binary
Actuator Ports
    port genPower type double
Input Message Ports
    port reqPower type double
Output Message Ports
    port status type binary
Task Input Ports
    port i1 type double
    port i2 type binary
Task Output Ports
    port o1 type double
    port o2 type binary
Task Private Ports

Tasks
    task t1 input i1 output o1 o2 function f1
    task t2 input i2 output o2 function f2

Drivers
    driver d1 source reqPower customerOverride
                guard g1 destination i1 i2 function h1
    driver d2 source o1 guard g2 destination genPower
                function h2
    driver d3 source o2 guard g3 destination status
                function h3
    driver d4 source o1 o2 guard g4
                destination o2 function h4
    driver d5 source customerOverride guard g5
                destination i2 function h5
    driver d6 source o2 guard g6
                destination o1 o2 function h6

Modes
    // Normal operating mode
    mode m1 period 10000ms ports i1 i2 o1 o2
      frequency 1 invoke task t1 driver d1
      frequency 1 update d2
      frequency 1 update d3
      frequency 1 switch m2 driver d4

    // User override mode
    mode m2 period 1000ms ports i2 o2
      frequency 1 invoke task t2 driver d5
      frequency 1 update d3
      frequency 2 switch m1 driver d6

Start m1
```

```
function f1( ) {
    o1 = i1;
    o2 = true;
}
function f2( ) {
    o2 = false;
}
function h1( ) {
    i1 = reqPower;
    i2 = customerOverride;
}
function h2( ) {
    genPower = o1;
}
...
...

binary guard g1( ) {
    return true;
}
...
...
binary guard g4( ) {
    return customerOverride;
}
binary guard g5( ) {
    return true;
}
binary guard g6( ) {
    return !customerOverride;
}
```

[a] Some guard and driver functions have been omitted to avoid unnecessary details.

CHAPTER IX

SERVICE-BASED COMPUTING PLATFORM

According to the proposed CPS reference model, once a CPS design has been decomposed into a set of interacting services, each with its own service description, these services are then deployed on various computing nodes that are involved in the CPS application. To enable CPS development according to the proposed CPS reference model, each CPS computing node must have an appropriate service-based computing platform that can support resource-aware service deployment and QoS-aware service operation. Figure 9.1 shows the role played by a service-based CPS computing platform in the context of the proposed service-oriented CPS reference model.

9.1 Requirements

Generally, a CPS scenario involves a set of heterogeneous computing nodes with different processors, operating systems, and middleware technologies. Therefore, the required service-based computing platform must be capable of being ported to these heterogeneous computing nodes. Moreover, resource-aware deployment of a service on a computing platform, as suggested by the proposed reference model, requires the existence of an appropriate service compiler as a part of the service-based computing platform. This service compiler must be capable of reading the service description (specified using an appropriate service description language) and deciding whether a certain computing nodes has enough resources to successfully deploy this service such that the service can meet its QoS constraints.

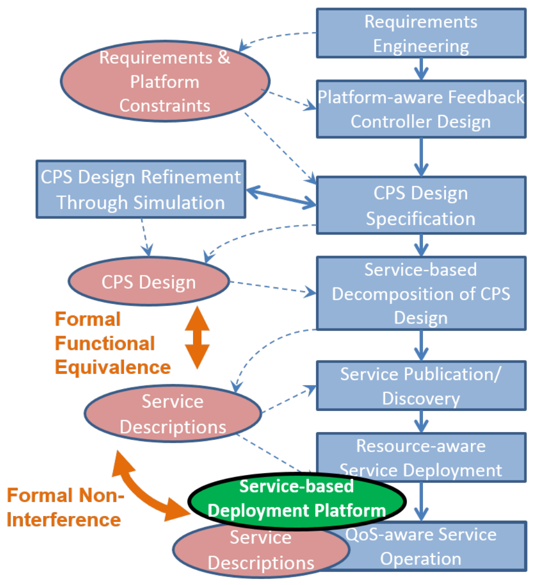

Requirements Engineering
Requirements & Platform Constraints
Platform-aware Feedback Controller Design
CPS Design Refinement Through Simulation
CPS Design Specification
CPS Design
Service-based Decomposition of CPS Design
Formal Functional Equivalence
Service Publication/ Discovery
Service Descriptions
Resource-aware Service Deployment
Formal Non-Interference
Service-based Deployment Platform
Service Descriptions
QoS-aware Service Operation

Figure 9.1: Role of the service-based computing platform (presented in this chapter) in the service-oriented CPS reference model.

9.2 Design

This section presents the design of a service-based computing platform for CPS computing nodes that is capable of supporting resource-aware deployment and QoS-aware operation of services whose service descriptions have been developed using the service description language (SDL), proposed in Chapter 8.

9.2.1 Embedded Machine (E Machine)

Section 8.2.1 had summarized various aspects of Giotto, a platform-independent programming language for embedded control systems. In real-time systems literature, development of Giotto compilers for various computing platforms has been reported [25]. However, while developing these Giotto compilers, researchers have found it useful to have an intermediate language, which does not support the high-level concepts of Giotto but still provides a lower level platform-independent semantics for mediating between physical environment and software tasks [26]. The concept of such an intermediate language has evolved into *E code*. Moreover, in the literature, the term *Embedded Machine* or *E Machine* has been used for a virtual machine that interprets

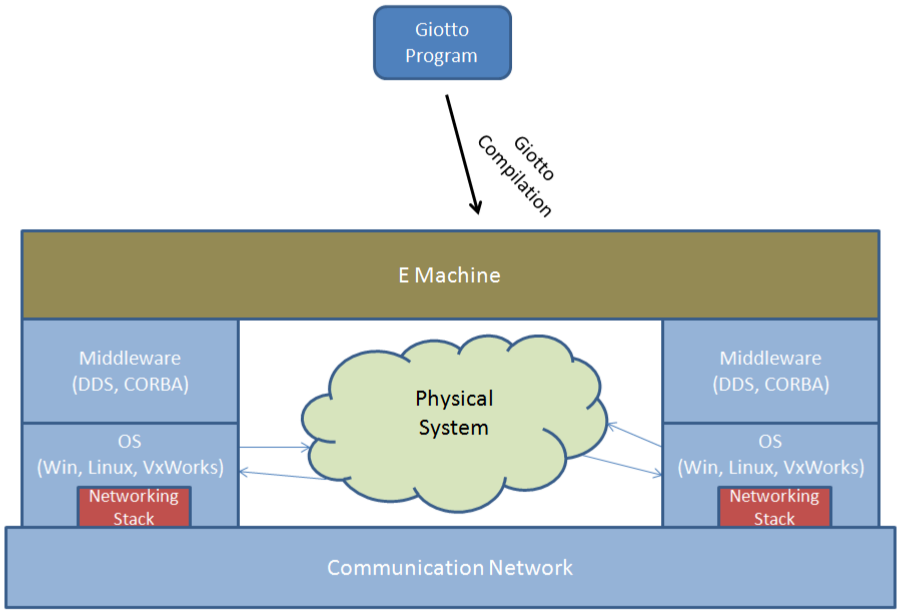


Figure 9.2: Typical configuration of Giotto and E Machine for embedded control systems.

the *E code* [26]. Figure 9.2 shows the Giotto and E Machine configuration for a typical networked embedded control system.

The proposed *E code* essentially has the following three instructions:

1. *Call driver*

2. *Release task*

3. *Future E code*

In the *E Code* terminology, a *task* is a piece of application-level code, whose execution takes non-zero time. When invoked with its parameters, a *task* implements a computational activity and writes the results to *task* ports. On the other hand, a *driver* is a piece of system-level code that typically enables a communication activity. For example, a *driver* can provide sensor readings as arguments to a *task* or load *task* results from its ports to an actuator. It is assumed that the execution of a *driver* takes logically zero time.

Call driver instruction starts the execution of a *driver*. As the *driver* is supposed to execute in logically zero time, the *E Machine* waits until the driver completes execution before interpreting the next instruction of E code. *Release task* instruction hands off a task to the operating system. Typically, the task is put into the ready queue of the operating system. Scheduler of the operating system is not under the control of the *E Machine*. The scheduler may or may not be able to satisfy the real-time constraints of the *E code*. However, a compiler (which takes into account the platform resources) checks the time safety of *E code*, generated from a higher level language, such as Giotto. Such a compiler attempts to rule out any timing violations by knowing the worst-case execution time (WCET) of all the tasks and by applying the schedulability results available in the real-time systems literature [10].

Future E code instruction marks a block of *E code* for execution at some future time. This instruction has two parameters: a trigger and the address of the block of *E code*. The trigger is evaluated with every input event (such as clock, sensor, or task output) and the block of *E code* is executed as soon as the trigger evaluates to true.

9.2.2 Combination of Embedded Machine (E Machine) and Compiler Machine (C Machine)

Since *E Machine*, summarized in the last section, supports resource-aware deployment and QoS-aware execution of Giotto programs, and a Giotto-based service description language has already been proposed in Chapter 8, it is natural to leverage *E Machine* as the foundation of required service-based computing platform. However, as noted in the last section, *E code* must be generated by an appropriate compiler to ensure time safety. Therefore, the required service-based computing platform must combine the *E Machine* with an appropriate service compiler that ensures resource-aware service deployment on *E Machine*. However, the service compiler code itself is not hard real-time in nature. Therefore, the proposed design of the service-based computing platform is based on splitting the resources of host computing platform into two

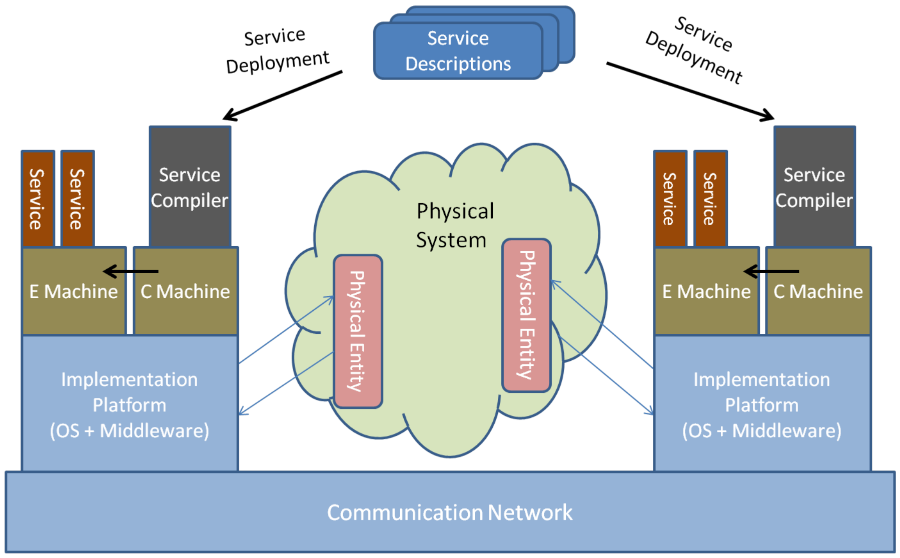


Figure 9.3: Proposed solution for the requirement of a service-based computing platform.

"virtual machines": a hard real-time Embedded Machine (E Machine) and a soft real-time Compiler Machine (C Machine). E Machine executes the hard real-time service code and C Machine executes the soft real-time code for service compiler. Resources of the host computing platform can be split into the hard real-time E Machine and soft real-time C Machine using various resource reservation schemes, reported in real-time systems literature [34] [35]. The resulting service-based computing platform is shown in Figure 9.3.

9.3 Case Study

This section shows the role played by the proposed service-based computing platform in the context of smart grid case study, presented in Chapter 5. Demand response application, involved in the smart grid case study of Chapter 5, consists of three computing nodes: CompNodeA, CompNodeB, and CommandCenter. Moreover, as discussed in Chapter 8, demand response application design can be decomposed into three services: *DemandResponseServiceA*, *DemandResponseServiceB*, and *DemandResponseServiceCC*. Figure 9.4 shows the cyber subsystem of demand response

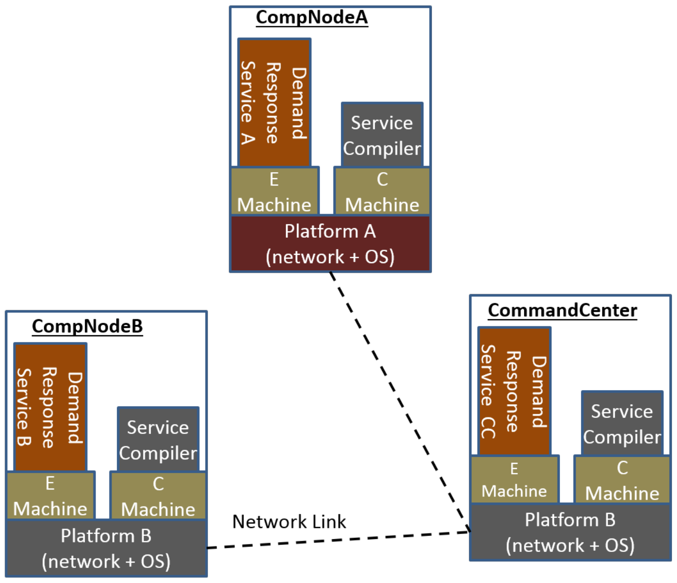

CompNodeA
Demand Response Service A
Service Compiler
E Machine
C Machine
Platform A (network + OS)
CompNodeB
Demand Response Service B
Service Compiler
E Machine
C Machine
Platform B (network + OS)
Network Link
CommandCenter
Demand Response Service CC
Service Compiler
E Machine
C Machine
Platform B (network + OS)

Figure 9.4: Case study: demand response application with proposed service-based computing platform.

application from smart grid case study, where a service-based computing platform (consisting of a combination of E Machine and C Machine) has been ported onto each of the computing nodes and the appropriate service has been deployed on that computing node through the service compiler component of the proposed computing platform.

CHAPTER X

FORMAL PERFORMANCE GUARANTEES

As discussed in Chapter 3, the proposed service-oriented CPS reference model requires the existence of formal guarantees for the following aspects:

1. functional equivalence between a CPS design specification and the corresponding service-based CPS deployment.
2. non-interference between the co-deployed CPS services from the perspective of their timing performance.

Using state-of-the-art techniques from the field of formal methods for reactive computer systems, this chapter shows how the technological solutions, presented in last four chapters (Chapter 6, Chapter 7, Chapter 8, and Chapter 9), combine to provide the above mentioned formal performance guarantees.

10.1 Formal Methods: A Short Introduction

The field of *formal methods* deals with techniques that guarantee the behavior of a computing system using some rigorous approach. Figure 10.1 summarizes the basic framework that is shared by various techniques, grouped under the umbrella of *formal methods* [2]. Typically, a computer systems is represented in terms of a specification formalism or an implementation construct (such as a programming language). A correctness property of this computer system is described as a formula of a mathematical logic system (such as propositional logic, first-order logic or temporal logic) [5]. Then, during the formal verification step, it is checked whether the correctness property holds for this computer system. There are two main approaches to the formal verification step: *model checking* and *deductive verification*. In *model checking* approach,

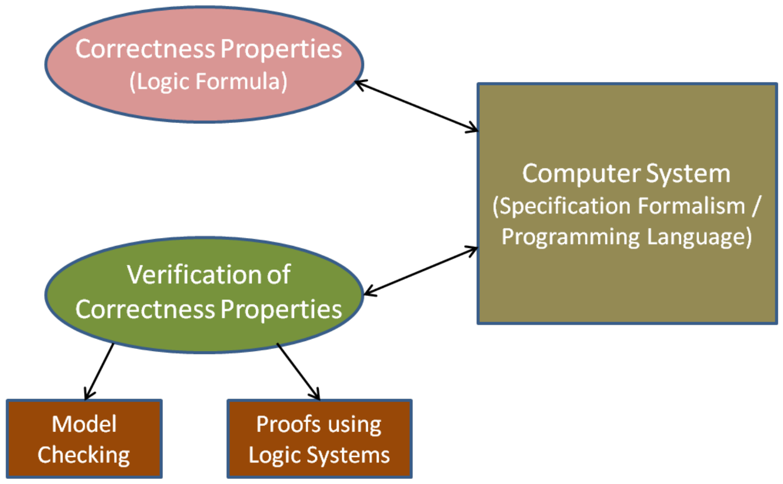

Correctness Properties
(Logic Formula)
Computer System
(Specification Formalism / Programming Language)
Verification of Correctness Properties
Model Checking
Proofs using Logic Systems

Figure 10.1: Basic framework employed by the field of formal methods.

all the states of a computer systems are traversed and the existence of correctness property is checked in each of the state. In *deductive verification* approach, a formal proof is developed for the existence of correctness property in each state of the computer system using a mathematical logic system (such as propositional logic) [2].

Computer systems can be classified into two main groups: *sequential computer systems* and *reactive computer systems*. Sequential computer systems enter a computation with a set of inputs, step through a set of instructions that represent the computation, and exit this computation with a set of outputs. On the other hand, reactive computer systems are characterized by an on-going interaction with their environment. In the field of formal methods, vastly different techniques are employed for these two different types of computer systems [5].

10.2 Formal Methods for Reactive Computer Systems: Manna-Pnueli Approach

In their seminal work on the application of linear temporal logic (LTL) for formal verification of reactive computer systems, Manna and Pnueli [46] [47] 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

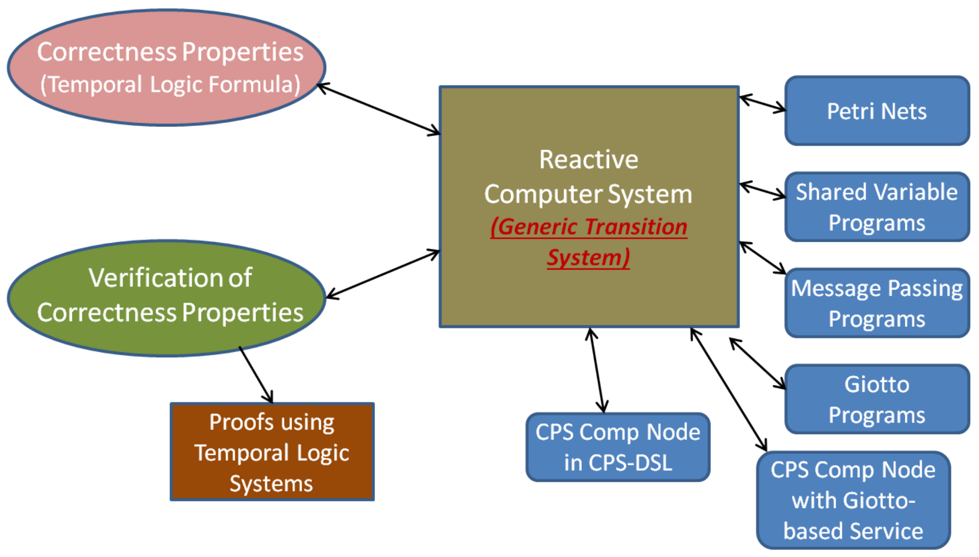

Correctness Properties (Temporal Logic Formula)
Verification of Correctness Properties
Proofs using Temporal Logic Systems
Reactive Computer System
(Generic Transition System)
Petri Nets
Shared Variable Programs
Message Passing Programs
Giotto Programs
CPS Comp Node in CPS-DSL
CPS Comp Node with Giotto-based Service

Figure 10.2: Formal methods for reactive computer systems: Manna-Pnueli approach.

dissertation.) 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. It must be noted that Giotto-based CPS services descriptions (proposed in Chapter 8) and cyber system elements of CPS-DSL (proposed in Chapter 6) are essentially two newly proposed representations of reactive computer systems. Formal proofs, presented in this chapter, leverage the decomposition of these newly proposed reactive computer system representations into the generic model of a *Manna-Pnueli Transition System*.

10.2.1 Manna-Pnueli Transition System

Manna-Pneuli 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 $\tau : \Sigma \to 2^{\Sigma}$ 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 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.

10.2.2 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.

10.2.3 Behavioral Equivalence

In the study and analysis of reactive computer systems, an important concept is the notion of behavioral equivalence between two different systems. Based on the transition-system-based generic model of reactive computer systems, proposed by Manna and Pnueli [46], one may try to define two transition systems P and P' to be *equivalent* if they generate precisely the same set of computations. However, as noted by Manna and Pnueli [46], this definition of equivalence will be too discriminating. There are many cases of reactive computer system programs that generate different computations, but still have equivalent behavior with respect to the outputs of interest. Motivated by this, Manna and Pnueli defined the following concepts about behavioral equivalence of reactive computer systems in their seminal work:

- O — *Observable Variables*

A subset of the state variables Π may be defined as *observable variables*, denoted by O. So, by definition, $O \subseteq \Pi$.

- $s \restriction O$ — *Observable State*

  Given a state s, *observable state* corresponding to s, denoted by $s \restriction O$, is defined as the restriction of s to just the observable variables O.

- σ^O — *Observable Behavior*

  Given a computation

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

  the *observable behavior* σ^O corresponding to σ is defined to be the sequence obtained from σ by replacing each state s_i with its corresponding observable state $s_i \restriction O$.

  $$\sigma^O : s_0 \restriction O, s_1 \restriction O, \ldots$$

- σ^r — *Reduced Behavior*

  Given a computation

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

  the *reduced behavior* σ^r corresponding to σ is defined to be the sequence obtained from σ by the following two transformations:

  1. Replace each state s_i by its observable part $s_i \restriction O$.
  2. Omit from the sequence each observable state that is identical to its predecessor but not identical to all of its successors.

- $\sim$ — *Equivalence of Transition Systems*

  For a transition system P, $R(P)$ denotes the set of all reduced behaviors generated by P. Let P_1 and P_2 be two transition systems and $O \subseteq \Pi_1 \subseteq \Pi_2$ be a set of variables, specified to be the observable variables for both systems.

The transition systems P_1 and P_2 are defined to be *equivalent* (relative to O), denoted by

$$P_1 \sim P_2$$

if $R(P_1) = R(P_2)$.

10.3 Proposed Extensions to Manna-Pnueli Approach

In order to utilize Manna-Pnueli Transition System for formal proofs about the proposed service-oriented CPS technologies, some new concepts must be defined:

- σ^{tr} — *Temporally Reduced Behavior*

  If one of the observable variables is *time*, then given a computation

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

  the *temporally reduced behavior* σ^{tr} corresponding to σ is defined to be the sequence obtained from σ by the following four transformations:

  1. Replace each state s_i by its observable part $s_i \restriction O$.
  2. Omit from the sequence each observable state that is identical to its predecessor but not identical to all of its successors.
  3. Omit from the sequence each observable state that is identical to its predecessor for all the observable variables except *time*.
  4. Omit from the sequence each observable state which has the same value of observable variable *time* as its successor.

  Based on this definition, the temporally reduced behavior of transition system P under the set of observable variables O ($\sigma_P^{tr} \restriction O$) is a sequence of tuples that captures the value of *time* as well as every other observable state variable in O at all the time instants at which the value of at least one non-time observable state variable from O changes. However, this sequence does not contain any two entries with the same value of state variable *time*.

The relationship of the temporally reduced behavior of P under O ($\sigma_P^{tr} \upharpoonright O$) and the elements of transitions associated with P can also be represented as follows:

$$\sigma_P^{tr} \upharpoonright O = f(\Delta_{T_P^{relevant} \upharpoonright O}, TimedTransitionSequence_{T_P^{relevant} \upharpoonright O})$$

where

$$\begin{aligned}
T_P^{relevant} \upharpoonright O = \{\tau_i \mid (\tau_i \in T_P) \wedge (&\text{modification statements of transition } \tau_i \\
&\text{change the value of non-time state variables in } O)\}
\end{aligned}$$

$$\Delta_{T_P^{relevant} \upharpoonright O} = \{\Delta_{\tau_i} \mid \tau_i \in T_P^{relevant} \upharpoonright O\}$$

$$\Delta_{\tau_i} = \{\Delta s_{\tau_i} \mid s \text{ is a non-time state variable in } O\}$$

$$\begin{aligned}
\Delta s_{\tau_i} = s'_{\tau_i} - s_{\tau_i} = {} &\text{Change in the value of state variable } s\text{, caused} \\
&\text{by transition } \tau_i
\end{aligned}$$

$$\begin{aligned}
TimedTransitionSequence_{T_P^{relevant} \upharpoonright O} = {} &(t_0, \tau_0), (t_1, \tau_1), (t_2, \tau_2), \ldots \\
&\text{such that:} \\
&\text{1) for each element } (t_i, \tau_i), \tau_i \in T_P^{relevant} \upharpoonright O \text{ and} \\
&\text{system reaches } time\ t_i \text{ after transition } \tau_i \text{ is taken.} \\
&\text{2) } t_{i+1} \geq t_i
\end{aligned}$$

- $\sim$ — *Equivalence of Transition Systems*

For a transition system P with *time* as an observable variable, $RT(P)$ denotes the set of all temporally reduced behaviors generated by P. Let P_1 and P_2 be two transition systems and $O \subseteq \Pi_1 \subseteq \Pi_2$ be a set of variables, specified to be the observable variables for both systems. The transition systems P_1 and P_2 are defined to be *equivalent* (relative to O), denoted by

$$P_1 \sim P_2$$

if $RT(P_1) = RT(P_2)$.

10.4 Manna-Pnueli Transition System Representation: CPS Computing Node in CPS-DSL

According to the CPS design specification language (CPS-DSL), proposed in Chapter 6, 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 in Appendix A, where:

- $\Pi_{P_{CompNode}}$ — Set of *state variables* of $P_{CompNode}$.
- $\Sigma_{P_{CompNode}}$ — Set of *states* of $P_{CompNode}$.
- $T_{P_{CompNode}}$ — Set of *transitions* of $P_{CompNode}$.
- $\Theta_{P_{CompNode}}$ — *Initial condition* of $P_{CompNode}$.

10.5 Manna-Pnueli Transition System Representation: CPS Computing Node with 1 CPS Service

A Giotto-based service description language (SDL) and a service-based CPS computing platform have been proposed in Chapter 8 and Chapter 9 respectively. Based on these proposed technologies, a CPS computing node with one successfully deployed Giotto-based CPS service, $Service1$, can be represented as the Manna-Pnueli Transition System, $P_{1Service} < \Pi_{P_{1Service}}, \Sigma_{P_{1Service}}, T_{P_{1Service}}, \Theta_{P_{1Service}} >$, outlined in Appendix B, where:

- $\Pi_{P_{1Service}}$ — Set of *state variables* of $P_{1Service}$.
- $\Sigma_{P_{1Service}}$ — Set of *states* of $P_{1Service}$.

- $T_{P_{1Service}}$ — Set of *transitions* of $P_{1Service}$.

- $\Theta_{P_{1Service}}$ — *Initial condition* of $P_{1Service}$.

10.6 Manna-Pnueli Transition System Representation: CPS Computing Node with k CPS Services

A Giotto-based service description language (SDL) and a service-based CPS computing platform have been proposed in Chapter 8 and Chapter 9 respectively. Based on these proposed technologies, a CPS computing node with k successfully deployed Giotto-based CPS services ($Service1, Service2, \ldots, ServiceK$) can be represented as the Manna-Pnueli Transition System, $P_{kServices} < \Pi_{P_{kServices}}, \Sigma_{P_{kServices}}, T_{P_{kServices}}, \Theta_{P_{kServices}} >$, outlined in Appendix C, where:

- $\Pi_{P_{kServices}}$ — Set of *state variables* of $P_{kServices}$.

- $\Sigma_{P_{kServices}}$ — Set of *states* of $P_{kServices}$.

- $T_{P_{kServices}}$ — Set of *transitions* of $P_{kServices}$.

- $\Theta_{P_{kServices}}$ — *Initial condition* of $P_{kServices}$.

10.7 Functional Equivalence of CPS Design Specification and Service-based CPS Deployment

First formal guarantee, required by the proposed CPS reference model, is the functional equivalence between a CPS design specification and the corresponding service-based CPS deployment. This dissertation has presented a CPS Design Specification Language (CPS-DSL) and a Giotto-based Service Description Language (SDL) in Chapter 6 and Chapter 8 respectively. Since *ComputingNode* block of CPS-DSL (proposed in Chapter 6) and a CPS computing node with a successfully deployed Giotto-based CPS service (proposed in Chapter 8) are essentially two newly proposed representations of reactive computer systems, these newly proposed representations can be translated into corresponding Manna-Pnueli Transition Systems (which was designed as a generic model of reactive computer systems). Appendix A and Appendix B provide the Manna-Pnueli Transition System representation for a *ComputingNode* block ($CompNode1$) of CPS-DSL and a CPS computing node with one Giotto-based service ($Service1$) respectively. Furthermore, using the notation from Appendix A and Appendix B, following properties must hold by design between a *ComputingNode* block of CPS-DSL ($CompNode1$) and the service description of the corresponding Giotto-based CPS service ($Service1$).

1. $f_{ModesMap}$ is a bijective function,

   where $f_{ModesMap}\colon Modes_{CompNode1} \rightarrow Modes_{Service1}$ is defined as:

   $mode^{i}_{Service1} = f_{ModesMap}(mode^{i}_{CompNode1})$

2. $f_{ModeSwitchesMap}$ is a bijective function,

   where $f_{ModeSwitchesMap}\colon ModeSwitches_{CompNode1} \rightarrow ModeSwitches_{Service1}$ is defined as:

   $modeSwitch^{ij}_{Service1} = f_{ModeSwitchesMap}(modeSwitch^{ij}_{CompNode1})$

3. $f_{SenosrPortsMap}$ is a bijective function,

   where $f_{SensorPortsMap}\colon SensorPorts_{CompNode1} \rightarrow SensorPorts_{Service1}$ is defined as:

   $sensePort^{i}_{Service1} = f_{SensorPortsMap}(sensePort^{i}_{CompNode1})$

   Similarly defined functions $f_{InMsgPortsMap}$, $f_{ActuatorPortsMap}$, and $f_{OutMsgPortsMap}$ are also bijective functions.

4. $f_{ControllerTasksMap}$ is a bijective function,

   where $f_{ControllerTasksMap}\colon ControllerFunctions_{CompNode1} \rightarrow Tasks_{Service1}$ is defined as:

$task^{i}_{Service1} = f_{ControllerTasksMap}(controllerFunction^{i}_{CompNode1})$

5. $f^{mode_i}_{SenosrPortsMap}$ is a bijective function,

   where $f^{mode_i}_{SensorPortsMap}\colon SensorPorts^{mode_i}_{CompNode1} \rightarrow SensorPorts^{mode_i}_{Service1}$ is defined as:

   $sensePort^{i}_{Service1} = f^{mode_i}_{SensorPortsMap}(sensePort^{i}_{CompNode1})$

   Similarly defined functions $f^{mode_i}_{InMsgPortsMap}$, $f^{mode_i}_{ActuatorPortsMap}$, and $f^{mode_i}_{OutMsgPortsMap}$ are also bijective functions.

6. $f^{mode_i}_{ControllerTasksMap}$ is a bijective function,

   where $f^{mode_i}_{ControllerTasksMap}\colon ControllerFunctions^{mode_i}_{CompNode1} \rightarrow Tasks^{mode_i}_{Service1}$ is defined as:

   $task^{i}_{Service1} = f^{mode_i}_{ControllerTasksMap}(controllerFunction^{i}_{CompNode1})$

7. $\forall\ mode_i \in Modes_{CompNode1}$

   $Period_{mode_j} = Period_{mode_i}$

   where $mode_j = f_{ModesMap(mode_i)}$

8. $\forall\ modeSwitch_i \in ModesSwitches_{CompNode1}$

   $SwitchFreq_{modeSwitch_j} = SwitchFreq_{modeSwitch_i}$

   where $modeSwitch_j = f_{ModeSwitchesMap(modeSwitch_i)}$

9. $\forall\ controllerFunction_i \in ControllerFunctions_{CompNode1}$

   $TaskFreq_{task_j} = ControllreFunctionFreq_{controllerFunction_i}$

   where $task_j = f_{ControllerTasksMap(controllerFunction_i)}$

10. $\forall\ controllerFunction_i \in ControllerFunctions_{CompNode1}$

    $f_{task_j} = f_{controllerFunction_i}$

where $task_j = f_{ControllerTasksMap(controllerFunction_i)}$

$f^{controllerFunction_i}$ = The function implemented by the internal components (Simulink blocks) of *ControllerFunction* block $controllerFucntion_i$

f^{task_j} = The function implemented in $task_j$ of CPS service $Service1$

11. $ControllerOutsToActs^{mode_i}_{CompNode1} = TaskOutsToActs^{mode_i}_{Service1}$

where

$ControllerOutsToActs^{mode_i}_{CompNode1}$:

$PeriodicControllerOutputValues^{mode_i}_{CompNode1} \rightarrow ActPortValues^{mode_i}_{CompNode1}$

= A function that captures the input-output relationship (produced by the combined effect) of all the connections between *PeriodicControllerOutput* blocks and *ActuatorPort* blocks in $mode_i$ of $CompNode1$.

$TaskOutsToActs^{mode_i}_{Service1}$:

$TaskOutputPortValues^{mode_i}_{Service1} \rightarrow ActPortValues^{mode_i}_{Service1}$

= A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the actuator ports in $mode_i$ of CPS service $Service1$.

12. $ControllerOutsToOutMsgs^{mode_i}_{CompNode1} = TaskOutsToOutMsgs^{mode_i}_{Service1}$

where

$ControllerOutsToOutMsgs^{mode_i}_{CompNode1}$:

$PeriodicControllerOutputValues^{mode_i}_{CompNode1} \rightarrow OutMsgPortValues^{mode_i}_{CompNode1}$

= A function that captures the input-output relationship (produced by the combined effect) of all the connections between *PeriodicControllerOutput* blocks and *OutputMsgPort* blocks in $mode_i$ of $CompNode1$.

$TaskOutsToOutMsgs^{mode_i}_{Service1}$:

$TaskOutputPortValues^{mode_i}_{Service1} \rightarrow OutputMsgPortValues^{mode_i}_{Service1}$

= A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the output message ports in $mode_i$ of CPS service $Service1$.

13. $\forall\ controllerFunction_j \in mode^i_{CompNode1}$, and

when $task_j = f_{ControllerTasksMap}(controllerFunction_j)$

$$LoadControllerInputs^{mode_i}_{controllerFunction_j} = LoadTaskInputs^{mode_i}_{task_j}$$

where

$LoadContrllerInputs^{mode_i}_{controllerFunction_j}$:

$\{SensorPortValues^{mode_i}_{CompNode1} \cup InputMsgPortValues^{mode_i}_{CompNode1}$

$\cup\ PeriodicControllerOutputValues^{mode_i}_{CompNode1}\}$

$\rightarrow PeriodicControllerInputValues_{controllerFunction_j}$

= A function that captures the input-output relationship (produced by the combined effect) of all the connections from *SensorPort*, *InMsgPort*, and *PeriodicControllerOutput* blocks in $mode_i$ of $CompNode1$ to the *PeriodicControllerInput* blocks, associated with *ControllerFunction* block $controllerFunction_j$ in $mode_i$ of $CompNode1$.

$LoadTaskInputs^{mode_i}_{task_j}$:

$\{SensorPortValues^{mode_i}_{Service1} \cup InputMsgPortValues^{mode_i}_{Service1}$

$\cup\ TaskOutputPortValues^{mode_i}_{Service1}\} \rightarrow TaskInputPortValues_{task_j}$

= A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the task input ports of $task_j$ in $mode_i$ of CPS service $Service1$.

14. $ModeSwitchFunction_{CompNode1}^{mode_i mode_j} = ModeSwitchFunction_{Service1}^{mode_i mode_j}$

where

$ModeSwitchFunction_{CompNode1}^{mode_i mode_j} : PeriodicControllerOutputValues_{CompNode1}^{mode_i}$

$\rightarrow PeriodicControllerOutputValues_{CompNode1}^{mode_j}$

= A function that takes as input the values of $periodicControllerOutput$ blocks in $mode_i$ and produces the values to which $periodicControllerOutput$ blocks in $mode_j$ are initialized after the *mode switch* from $mode_i$ to $mode_j$ of *ControlApp*, associated with $CompNode1$.

$ModeSwitchFunction_{Service1}^{mode_i mode_j} : TaskOutputValues_{Service1}^{mode_i}$

$\rightarrow TaskOutputValues_{Service1}^{mode_j}$

= The function used in the definition of the *driver* associated with the *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service1$

The formal guarantee of equivalence between a CPS design specification and the corresponding service-based CPS deployment can be stated in terms of Manna-Pnueli Transition System for reactive computer systems by the following theorem:

Theorem 10.1. *Let $CompNode1$ be a ComputingNode block in a CPS design specification, and let $Service1$ be the corresponding CPS service in the service-based CPS deployment. Given*

$P_{CompNode}$ = *Manna-Pnueli Transition System representation of ComputingNode block, $CompNode1$, in a CPS design specification*

$P_{1Service}$ = *Manna-Pnueli Transition System representation of a CPS computing node with one successfully deployed Giotto-based CPS service, Service*1,

$O_{in}^{CompNode}$ = *Time* ∪ *{Set of SensorPort blocks, contained in CompNode*1*}* ∪ *{Set of InputMsgPort blocks, contained in CompNode1},*

$O_{in}^{Service}$ = *Time* ∪ *{Set of Sensor Ports for Service1}* ∪ *{Set of Input Message Ports for Service1},*

$O_{out}^{CompNode}$ = *Time* ∪ *{Set of ActuatorPort blocks, contained in CompNode*1*}* ∪ *{Set of OutputMsgPort blocks, contained in CompNode1},*

and

$O_{out}^{Service}$ = *Time* ∪ *{Set of Actuator Ports for Service1}* ∪ *{Set of Output Message Ports for Service1}.*

For arbitrary computations $\sigma_{P_{CompNode}}$ *and* $\sigma_{P_{Service}}$, *if*

$$\sigma^{tr}_{P_{CompNode}} \textit{ under } O_{in}^{CompNode} = \sigma^{tr}_{P_{1Service}} \textit{ under } O_{in}^{Service}$$

then

$$\sigma^{tr}_{P_{CompNode}} \textit{ under } O_{out}^{CompNode} = \sigma^{tr}_{P_{1Service}} \textit{ under } O_{out}^{Service}$$

Proof. As outlined in Section 10.3, the temporally reduced behavior of transition system P under observable variables O ($\sigma_P^{tr} \upharpoonright O$) can also be represented as follows:

$$\sigma_P^{tr} \upharpoonright O = f(\Delta_{T_P^{relevant} \upharpoonright O}, TimedTransitionSequence_{T_P^{relevant} \upharpoonright O}) \qquad (A)$$

where

$$T_P^{relevant} \upharpoonright O = \{\tau_i \mid (\tau_i \in T_P) \wedge (\text{modification statements of transition } \tau_i \text{ change the value of non-time state variables in } O)\}$$

$$\Delta_{T_P^{relevant} \upharpoonright O} = \{\Delta_{\tau_i} \mid \tau_i \in T_P^{relevant} \upharpoonright O\}$$

$$\Delta_{\tau_i} = \{\Delta s_{\tau_i} \mid s \text{ is a non-time state variable in } O\}$$

$$\Delta s_{\tau_i} = s'_{\tau_i} - s_{\tau_i} = \text{Change in the value of state variable } s\text{, caused by transition } \tau_i$$

$$TimedTransitionSequence_{T_P^{relevant} \upharpoonright O} = (t_0, \tau_0), (t_1, \tau_1), (t_2, \tau_2), \ldots$$

such that:

1) for each element $(t_i, \tau_i), \tau_i \in T_P^{relevant} \upharpoonright O$ and system reaches *time* t_i after transition τ_i is taken.

2) $t_{i+1} \geq t_i$

Specializing (A) for transition system $P_{CompNode}$ and observable variables $O_{out}^{CompNode}$:

$$\sigma_{P_{CompNode}}^{tr} \upharpoonright O_{out}^{CompNode} = f_1(\Delta_{T_{P_{CompNode}}^{relevant} \upharpoonright O_{out}^{CompNode}}, TimedTransitionSequence_{T_{P_{CompNode}}^{relevant} \upharpoonright O_{out}^{CompNode}}) \qquad (A1)$$

Specializing (A) for transition system $P_{1Service}$ and observable variables $O_{out}^{Service}$:

$$\sigma_{P_{1Service}}^{tr} \upharpoonright O_{out}^{Service} = f_1(\Delta_{T_{P_{1Service}}^{relevant} \upharpoonright O_{out}^{Service}}, TimedTransitionSequence_{T_{P_{1Service}}^{relevant} \upharpoonright O_{out}^{Service}}) \qquad (A2)$$

From the Manna-Pnueli Transition System representation $P_{CompNode}$, presented in Appendix A, it can be seen that

$$T_{P_{CompNode}}^{relevant} \upharpoonright O_{out}^{CompNode} = T_{CompNode1}^{ModeSwitches}|_{P_{CompNode}} \cup T_{CompNode1}^{TimeIncrement}|_{P_{CompNode}} \qquad (B1)$$

where

$T_{CompNode1}^{ModeSwitches}|_{P_{CompNode}}$ = Set of transitions $T_{CompNode1}^{ModeSwitches}$, as defined in $P_{CompNode}$

$T_{CompNode1}^{TimeIncrement}|_{P_{CompNode}}$ = Set of transitions $T_{CompNode1}^{TimeIncrement}$, as defined in $P_{CompNode}$

From the modification statements of transitions $T_{CompNode1}^{ModeSwitches}|_{P_{CompNode}}$ and $T_{CompNode1}^{TimeIncrement}|_{P_{CompNode}}$, outlined in Appendix A, it can be seen that for $x > 0$ and $y \geq 0$:

$$\big(actPorts_{CompNode1}(t), outMsgPorts_{CompNode1}(t)\big) = f_a\big(periodicControllerOuts_{CompNode1}(t)\big),$$

$$periodicControllerOuts_{CompNode1}(t) = f_b\big(periodicControllerIns_{CompNode1}(t - x)\big),$$

and

$$periodicControllerIns_{CompNode1}(t) = f_c\big(sensePorts_{CompNode1}(t-y),$$
$$inMsgPorts_{CompNode1}(t-y)\big).$$

Therefore,

$$\big(actPorts_{CompNode1}(t), outMsgPorts_{CompNode1}(t)\big) =$$
$$f_d\big(sensePorts_{CompNode1}(t-x), inMsgPorts_{CompNode1}(t-x)\big),$$

As a result,

$$\Delta_{T^{relevant}_{P_{CompNode}} \restriction O^{CompNode}_{out}} = f_2\big(t, sensePorts_{CompNode1}(t-x),$$
$$inMsgPorts_{CompNode1}(t-x)\big)$$

Now, by definition, the temporally reduced behavior of $P_{CompNode}$ under the set of observable variables $O^{CompNode}_{in}$ $(\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{in})$ captures the the time and new value of sensor ports $(sensePorts_{CompNode1})$ and input message ports $(inMsgPorts_{CompNode1})$ of $CompNode1$ at every change in the sensor port values and input message port value. Therefore,

$$\Delta_{T^{relevant}_{P_{CompNode}} \restriction O^{CompNode}_{out}} = f_3\big(\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{in}\big) \qquad (C1)$$

From the Manna-Pnueli Transition System representation $P_{1Service}$, outlined in Appendix B, it can be seen that

$$T^{relevant}_{P_{1Service}} \restriction O^{Service}_{out} = T^{ModeSwitches}_{Service1}|_{P_{1Service}} \cup T^{TimeIncrement}_{Service1}|_{P_{1Service}} \qquad (B2)$$

where

$T^{ModeSwitches}_{Service1}|_{P_{1Service}}$ = Set of transitions $T^{ModeSwitches}_{Service1}$, as defined in $P_{1Service}$

$T^{TimeIncrement}_{Service1}|_{P_{1Service}}$ = Set of transitions $T^{TimeIncrement}_{Service1}$, as defined in $P_{1Service}$

From the modification statements of transitions $T^{ModeSwitches}_{Service1}|_{P_{1Service}}$ and $T^{TimeIncrement}_{Service1}|_{P_{1Service}}$, outlined in Appendix B, it can be seen that for $x > 0$ and $y \geq 0$:

$$\big(actPorts_{Service1}(t), outMsgPorts_{Service1}(t)\big) = f'_a\big(taskOutPorts_{Service1}(t)\big),$$

$$taskOutPorts_{Service1}(t) = f'_b\big(taskInPorts_{Service1}(t-x)\big),$$

and

$$taskInPorts_{Service1}(t) = f'_c\big(sensePorts_{Service1}(t-y), inMsgPorts_{Service1}(t-y)\big).$$

Therefore,

$$\big(actPorts_{Service1}(t), outMsgPorts_{Service1}(t)\big) = \\ f'_d\big(sensePorts_{Service1}(t-x), inMsgPorts_{Service1}(t-x)\big),$$

As a result,

$$\Delta_{T^{relevant}_{P_{1Service}}\restriction O^{Service}_{out}} = f'_2\big(t, sensePorts_{Service1}(t-x), \\ inMsgPorts_{Service1}(t-x)\big)$$

Now, by definition, the temporally reduced behavior of $P_{1Service}$ under the set of observable variables $O^{Service}_{in}$ ($\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}$) captures the the time and new value of sensor ports ($sensePorts_{Service1}$) and input message ports ($inMsgPorts_{Service1}$) of $Service1$ at every change in the sensor port values and input message port value. Therefore,

$$\Delta_{T^{relevant}_{P_{1Service}}\restriction O^{Service}_{out}} = f'_3\big(\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}\big)$$

Based on the properties that must hold by design between the *ComputingNode* block $CompNode1$ and the corresponding CPS service $Service1$ (listed earlier in this section), functions employed in the corresponding modification statements of $T^{relevant}_{P_{CompNode}}$ and $T^{relevant}_{P_{1Service}}$ are equal to each other. Therefore, functions f'_a, f'_b, f'_c, f'_d, f'_d, f'_2, and f'_3 are equal to functions f_a, f_b, f_c, f_d, f_2, and f_3 respectively. Hence,

$$\Delta_{T^{relevant}_{P_{1Service}}\restriction O^{Service}_{out}} = f_3\big(\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}\big) \qquad (C2)$$

Furthermore, by the definition of $TimedTransitionSequence$ presented earlier in the proof:

$$TimedTransitionSequence_{T^{relevant}_{P}\restriction O} = f_4\big(t, EnablingConditions_{T^{relevant}_{P}\restriction O}(t), \\ NextTimes_{T^{relevant}_{P}\restriction O}(t)\big) \qquad (D)$$

where

$$EnablingConditions_{T^{relevant}_{P}\restriction O} = \{EnablingCondition_{\tau_i} \mid \tau_i \in T^{relevant}_{P} \restriction O\}$$

$$EnablingCondition_{\tau}(t) = \text{status (true/false) of the enabling condition} \\ \text{of transition } \tau \text{ at time instant } t$$

$$NextTimes_{T_P^{relevant}\restriction O} = \{NextTime_{\tau_i} \mid \tau_i \in T_P^{relevant} \restriction O\}$$

$NextTime_\tau(t) =$ The value of state variable *time* after transition τ is taken at *time* t

Specializing (D) for transition system $P_{CompNode}$ and observable variables $O_{out}^{CompNode}$:

$$TimedTransitionSequence_{T_{P_{CompNode}}^{relevant}\restriction O_{out}^{CompNode}} = f_4\big(t, EnablingConditions_{T_{P_{CompNode}}^{relevant}\restriction O_{out}^{CompNode}}(t), NextTimes_{T_{P_{CompNode}}^{relevant}\restriction O_{out}^{CompNode}}(t)\big) \qquad (D1)$$

Specializing (D) for transition system $P_{1Service}$ and observable variables $O_{out}^{Service}$:

$$TimedTransitionSequence_{T_{P_{1Service}}^{relevant}\restriction O_{out}^{Service}} = f_4\big(t, EnablingConditions_{T_{P_{1Service}}^{relevant}\restriction O_{out}^{Service}}(t), NextTimes_{T_{P_{1Service}}^{relevant}\restriction O_{out}^{Service}}(t)\big) \qquad (D2)$$

From the enabling conditions (outlined in Appendix A) of set of transitions described in $(B1)$, it can be seen that for $y \geq 0$

$$EnablingConditions_{T_{P_{CompNode}}^{relevant}\restriction O_{out}^{CompNode}}(t) = f_5\big(ModeSwitchCheckTimes_{CompNode1}, ModeSwitchConditions_{CompNode1}(t-y)\big) \qquad (E1)$$

where

$ModeSwitchCheckTimes_{CompNode1} =$ {Set of time instants (relative to last mode switch time) at which mode switch conditions are checked according to the *ModeSwitchLogic* block, contained in the *ComputingNode* block $CompNode1$}.

$ModeSwitchConditions_{CompNode1}(t) =$ {Set that contains the status at time t of all the mode switch assertions associated with *ModeSwitchLogic* block, contained in the *ComputingNode* block $CompNode1$}.

From the definition of $ModeSwitchCheckTime_{CompNode1}(t, t_{CompNode1}^{switch}, mode_i, mode_j)$, presented in Appendix A:

$$ModeSwithCheckTimes_{CompNode1} = f_e(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}) \qquad (F1)$$

where

$$ModePeriods_{CompNode1} = \{Period_{mode_i} \mid mode_i \in Modes_{CompNode1}\}$$
$$ModeSwitchFreqs_{CompNode1} = \{SwitchFreq_{mode_i mode_j} \mid \exists \text{ a mode switch}$$
$$\text{from } mode_i \text{ to } mode_j \text{ of } CompNode1\}$$

Since mode switch decisions of a *ModeSwitchLogic* block, contained in a *ComputingNode* block, are made based on the values of sensor ports, actuator ports, input message ports, and output message ports associated with a *ComputingNode* block,

$$ModeSwithConditions_{CompNode1}(t) = f_g\big(sensePorts_{CompNode1}(t),$$
$$inMsgPorts_{CompNode1}(t), actPorts_{CompNode1}(t), outMsgPorts_{CompNode1}(t)\big),$$

According to the modification statements of transitions of $P_{CompNode}$, for $x > 0$:

$$\big(actPorts_{CompNode1}(t), outMsgPorts_{CompNode1}(t)\big) =$$
$$f_h\big(sensePorts_{CompNode1}(t-x), inMsgPorts_{CompNode1}(t-x)\big)$$

Therefore,

$$ModeSwithConditions_{CompNode1}(t) = f_i\big(sensePorts_{CompNode1}(t),$$
$$inMsgPorts_{CompNode1}(t), sensePorts_{CompNode1}(t-x),$$
$$inMsgPorts_{CompNode1}(t-x)\big)$$

Now, by definition, the temporally reduced behavior of $P_{CompNode}$ under the set of observable variables $O_{in}^{CompNode}$ ($\sigma^{tr}_{P_{CompNode}} \restriction O_{in}^{CompNode}$) captures the the time and new value of sensor ports ($sensePorts_{CompNode1}$) and input message ports ($inMsgPorts_{CompNode1}$) of $CompNode1$ at every change in the sensor port values and input message port value. Therefore,

$$ModeSwithConditions_{CompNode1}(t) = f_j(\sigma^{tr}_{P_{CompNode}} \restriction O_{in}^{CompNode}) \qquad (G1)$$

Combining (E1), (F1), and (G1):

$$EnablingConditions_{T^{relevant}_{P_{CompNode}} \restriction O_{out}^{CompNode}}(t) = f_6\big(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}, \sigma^{tr}_{P_{CompNode}} \restriction O_{in}^{CompNode}\big) \qquad (H1)$$

From the definition of t_{jump} used in the modification statements (outlined in Appendix A) of set of transitions described in $(B1)$, it can be seen that:

$$NextTimes_{T^{relevant}_{P_{CompNode}} \upharpoonright O^{CompNode}_{out}}(t) = f_7\big(t^{switch}_{CompNode1}(t), ModePeriods_{CompNode1},$$
$$ControllerFunctionFreqs_{CompNode1}\big) \qquad (J1)$$

where

$t^{switch}_{CompNode1}(t) =$ Time of the last mode switch of $CompNode1$ when the system is at time t

$ControllerFunctionFreqs_{CompNode1} = \{ControllerFucntionFreq_{function_i} \mid fucntion_i \in ContollerFucntions_{CompNode1}\}$

For $y \geq 0$

$$t^{switch}_{CompNode1}(t) = f_k\big(ModeSwitchCheckTimes_{CompNode1},$$
$$ModeSwitchConditions_{CompNode1}(t-y)\big) \qquad (K1)$$

Combining (F1), (G1), (J1), and (K1):

$$NextTimes_{T^{relevant}_{P_{CompNode}} \upharpoonright O^{CompNode}_{out}}(t) = f_8\big(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}, ControllerFunctionFreqs_{CompNode1},$$
$$\sigma^{tr}_{P_{CompNode}} \upharpoonright O^{CompNode}_{in}\big) \qquad (L1)$$

Combining (D1), (H1), and (L1):

$$TimedTransitionSequence_{T^{relevant}_{P_{CompNode}} \upharpoonright O^{CompNode}_{out}} = f_8\big(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}, ControllerFunctionFreqs_{CompNode1},$$
$$\sigma^{tr}_{P_{CompNode}} \upharpoonright O^{CompNode}_{in}\big) \qquad (M1)$$

In the next segment of the proof, the process of conversion between the two representations of $TimedTransitionSequence_{T^{relevant}_{P_{CompNode}} \upharpoonright O^{CompNode}_{out}}$, shown in (D1) and (M1), will be repeated for $TimedTransitionSequence_{T^{relevant}_{P_{1Service}} \upharpoonright O^{Service}_{out}}$.

From the enabling conditions (outlined in Appendix B) of set of transitions described in $(B2)$, it can be seen that for $y \geq 0$

$$EnablingConditions_{T^{relevant}_{P_{1Service}} \upharpoonright O^{Service}_{out}}(t) = f_5\big(ModeSwitchCheckTimes_{Service1},$$
$$ModeSwitchConditions_{Service1}(t-y)\big) \qquad (E2)$$

where

$ModeSwitchCheckTimes_{Service1} = \{$Set of time instants (relative to last mode switch time) at which mode switch conditions are checked according to the service description of CPS service $Service1\}$.

$ModeSwitchConditions_{Service1}(t) = \{$Set that contains the status at time t of all the mode switch assertions associated with CPS service $Service1\}$.

From the definition of $ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j)$, presented in Appendix B:

$$ModeSwithCheckTimes_{Service1} = f_e(ModePeriods_{Service1}, ModeSwitchFreqs_{Service1}) \qquad (F2)$$

where

$ModePeriods_{Service1} = \{Period_{mode_i} \mid mode_i \in Modes_{Service1}\}$

$ModeSwitchFreqs_{Service1} = \{SwitchFreq_{mode_i mode_j} \mid \exists$ a mode switch from $mode_i$ to $mode_j$ of $Service1\}$

Since mode switch decisions of a CPS service are made based on the values of sensor ports, actuator ports, input message ports, and output message ports of CPS service,

$$ModeSwithConditions_{Service1}(t) = f'_g\big(sensePorts_{Service1}(t), inMsgPorts_{Service1}(t), actPorts_{Service1}(t), outMsgPorts_{Service1}(t)\big),$$

According to the modification statements of transitions of $P_{1Service}$, for $x > 0$:

$$\big(actPorts_{Service1}(t), outMsgPorts_{Service1}(t)\big) = f'_h\big(sensePorts_{Service1}(t-x), inMsgPorts_{Service1}(t-x)\big)$$

Therefore,

$$ModeSwithConditions_{Service1}(t) = f'_i\big(sensePorts_{Service1}(t), inMsgPorts_{Service1}(t), sensePorts_{Service1}(t-x), inMsgPorts_{Service1}(t-x)\big)$$

Now, by definition, the temporally reduced behavior of $P_{1Service}$ under the set of observable variables $O_{in}^{Service}$ ($\sigma_{P_{1Service}}^{tr} \upharpoonright O_{in}^{Service}$) captures the the time and new value of sensor ports ($sensePorts_{Service1}$) and input message ports ($inMsgPorts_{Service1}$) of $Service1$ at every change in the sensor port values and input message port value. Therefore,

$$ModeSwithConditions_{Service1}(t) = f_j'(\sigma_{P_{1Service}}^{tr} \upharpoonright O_{in}^{Service}) \qquad (G2)$$

Based on the properties that must hold by design between the *ComputingNode* block $CompNode1$ and the corresponding CPS service $Service1$ (listed earlier in this section), functions employed in the corresponding modification statements of $T_{P_{CompNode}}^{relevant}$ and $T_{P_{1Service}}^{relevant}$ are equal to each other. Therefore, functions f_g', f_h', f_i', and f_j' are equal to functions f_g, f_h, f_i, and f_j respectively. Hence, combining (E2), (F2), and (G2):

$$EnablingConditions_{T_{P_{1Service}}^{relevant} \upharpoonright O_{out}^{Service}}(t) = f_6\big(ModePeriods_{Service1},$$
$$ModeSwitchFreqs_{Service1}, \sigma_{P_{1Service}}^{tr} \upharpoonright O_{in}^{Service}\big) \qquad (H2)$$

From the definition of t_{jump} used in the modification statements (outlined in Appendix B) of set of transitions described in $(B2)$, it can be seen that:

$$NextTimes_{T_{P_{1Service}}^{relevant} \upharpoonright O_{out}^{Service}}(t) = f_7\big(t_{Service1}^{switch}(t), ModePeriods_{Service1},$$
$$TaskFreqs_{Service1}\big) \qquad (J2)$$

where

$t_{Service1}^{switch}(t) =$ Time of the last mode switch of $Service1$ when the system is at time t

$TaskFreqs_{Service1} = \{TaskFreq_{task_i} \mid task_i \in Tasks_{Service1}\}$

For $y \geq 0$

$$t_{Service1}^{switch}(t) = f_k\big(ModeSwitchCheckTimes_{Service1},$$
$$ModeSwitchConditions_{Service1}(t-y)\big) \qquad (K2)$$

Combining (F2), (G2), (J2), and (K2):

$$NextTimes_{T_{P_{1Service}}^{relevant} \upharpoonright O_{out}^{Service}}(t) = f_8\big(ModePeriods_{Service1},$$
$$ModeSwitchFreqs_{Service1}, TaskFreqs_{Service1},$$

$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}) \qquad (L2)$$

Combining (D2), (H2), and (L2):

$$TimedTransitionSequence_{T^{relevant}_{P_{1Service}} \restriction O^{Service}_{out}} = f_8\big(ModePeriods_{Service1},$$
$$ModeSwitchFreqs_{Service1}, TaskFreqs_{Service1},$$
$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}\big) \qquad (M2)$$

Combining (A1), (C1), and (M1):

$$\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{out} = f_9(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}, ControllerFunctionFreqs_{CompNode1},$$
$$\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{in}) \qquad (N1)$$

Combining (A2), (C2), and (M2):

$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{out} = f_9(ModePeriods_{Service1},$$
$$ModeSwitchFreqs_{Service1}, TaskFreqs_{Service1},$$
$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}) \qquad (N2)$$

Based on the properties that must hold by design between the *ComputingNode* block $CompNode1$ and the corresponding CPS service $Service1$:

$$ModePeriods_{Service1} = ModePeriods_{CompNode1}$$

$$ModeSwitchFreqs_{Service1} = ModeSwitchFreqs_{CompNode1}$$

$$TaskFreqs_{Service1} = ControllerFunctionFreqs_{CompNode1}$$

Substituting these value in (N2)

$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{out} = f_9(ModePeriods_{CompNode1},$$
$$ModeSwitchFreqs_{CompNode1}, ControllerFunctionFreqs_{CompNode1},$$
$$\sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}) \qquad (N3)$$

By comparison of (N1) and (N3), it follows that if

$$\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{in} = \sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{in}$$

then

$$\sigma^{tr}_{P_{CompNode}} \restriction O^{CompNode}_{out} = \sigma^{tr}_{P_{1Service}} \restriction O^{Service}_{out}$$

□

10.8 Non-interference between Co-deployed CPS Services

Another formal guarantee, required by the proposed CPS reference model, is the non-interference between the co-deployed CPS services from the perspective of their timing performance. Since a CPS computing node with one or more successfully deployed Giotto-based CPS services is an example of a reactive computer system, it can be represented as a Manna-Pnueli Transition System (which was designed as a generic model for reactive computer systems). As a result, the formal guarantee of non-interference between co-deployed CPS services (from the perspective of their timing performance) can be stated in terms of Manna-Pnueli Transition System for reactive computer systems by the following theorem:

Theorem 10.2. *Given*

$P_{1Service}$ = *Manna-Pnueli Transition System representation of a CPS computing node with one successfully deployed Giotto-based CPS service, $Service1$,*

$P_{kServices}$ = *Manna-Pnueli Transition System representation of a CPS computing node with k successfully deployed Giotto-based CPS services that include $Service1$ and $k-1$ additional services,*

O_{in} = *{Set of observable variables}*

= *Time* $\cup$ *{Set of Sensor Ports for Service1}*

$\cup$ *{Set of Input Message Ports for Service1},*

and

O_{out} = *{Set of observable variables}*

= *Time* $\cup$ *{Set of Actuator Ports for Service1}*

$\cup$ *{Set of Output Message Ports for Service1}.*

For arbitrary computations $\sigma_{P_{1Service}}$ and $\sigma_{P_{kServices}}$, if

$$\sigma^{tr}_{P_{1Service}} \text{ under } O_{in} = \sigma^{tr}_{P_{kServices}} \text{ under } O_{in}$$

then

$$\sigma^{tr}_{P_{1Service}}\ under\ O_{out} = \sigma^{tr}_{P_{kServices}}\ under\ O_{out}$$

Proof. From the Manna-Pnueli Transition System Representation $P_{kServices}$, outlined in Appendix C, it can be seen that transitions of $P_{kServices}$ can be divided into the following disjoint subsets: $T^{Service1}_{P_{kServices}}, T^{Service2}_{P_{kServices}}, \ldots, T^{ServiceK}_{P_{kServices}}$. Therefore,

$$T_{P_{kServices}} = T^{Service1}_{P_{kServices}} \cup T^{Service2}_{P_{kServices}} \cup \cdots \cup T^{ServiceK}_{P_{kServices}}$$

where

$$\begin{aligned}
T^{Service1}_{P_{kServices}} &= T^{ModeSwitches}_{Service1}|_{P_{kServices}} \cup T^{TimeIncrement}_{Service1}|_{P_{kServices}} \\
&= \text{Set of transitions of } P_{kServices} \text{ that deal with CPS service } Service1 \\
T^{Service2}_{P_{kServices}} &= T^{ModeSwitches}_{Service2}|_{P_{kServices}} \cup T^{TimeIncrement}_{Service2}|_{P_{kServices}} \\
&= \text{Set of transitions of } P_{kServices} \text{ that deal with CPS service } Service2 \\
T^{ServiceK}_{P_{kServices}} &= T^{ModeSwitches}_{ServiceK}|_{P_{kServices}} \cup T^{TimeIncrement}_{ServiceK}|_{P_{kServices}} \\
&= \text{Set of transitions of } P_{kServices} \text{ that deal with CPS service } ServiceK
\end{aligned}$$

Moreover, based on the comparison of $P_{1Service}$ and $P_{kServices}$ (presented in Appendix B and Appendix C respectively), $f_{MapService1}$ is a bijective function, when $f_{MapService1}\colon T^{Service1}_{P_{1Service}} \to T^{Service1}_{P_{kServices}}$ is defined as:

$$f_{MapService1}(n) = \begin{cases} \tau^{mode_i}_{Service1}|_{P_{kServices}} & \text{if } n = \tau^{mode_i}_{Service1}|_{P_{1Service}} \\[2ex] \tau^{mode_i mode_j}_{Service1}|_{P_{kServices}} & \text{if } n = \tau^{mode_i mode_j}_{Service1}|_{P_{1Service}} \end{cases}$$

where

$\tau^{mode_i}_{Service1}|_{P_{1Service}} =$ Transition $\tau^{mode_i}_{Service1}$, as defined in $P_{1Service}$

$\tau^{mode_i}_{Service1}|_{P_{kServices}} =$ Transition $\tau^{mode_i}_{Service1}$, as defined in $P_{kServices}$

Before pursuing the proof of Theorem 10.2, proofs for some required lemmas are presented below:

From the enabling conditions of all the time-advancing transitions of $P_{kServices}$ ($\tau^{mode_i}_{Service1}|_{P_{kServices}}, \tau^{mode_i}_{Service2}|_{P_{kServices}}, \ldots, \tau^{mode_i}_{ServiceK}|_{P_{kServices}}$), it can be noticed that in a computation of $P_{kServices}$, a time-advancing transition $\tau^{mode_i}_{ServiceB}|_{P_{kServices}}$ is only taken at time t if

$t^{next}_{ServiceB}(t) = min\Big\{t^{next}_{Service1}(t), t^{next}_{Service2}(t), \ldots, t^{next}_{ServiceK}(t)\Big\}$

Furthermore, once the transition $\tau^{mode_i}_{ServiceB}|_{P_{kServices}}$ is taken at time t, its modification statements move the state variable *time* from t to $t' = t^{next}_{ServiceB}(t)$ and state variable $t^{next}_{ServiceB}$ to ${t^{next}_{ServiceB}}'\big(> t^{next}_{ServiceB}(t)\big)$. Therefore, in any computation of $P_{kServices}$, the following property always holds:

$$t \leq min\Big\{t^{next}_{Service1}(t), t^{next}_{Service2}(t), \ldots, t^{next}_{ServiceK}(t)\Big\}$$

From this property, it follows that.

In any computation of $P_{kServices}$, $t \leq t^{next}_{Service1}(t)$. ***(Lemma I)***

From the transitions of $P_{1Service}$ (outlined in Appendix B), it can be seen that state variable $t^{switch}_{Service1}|_{P_{1Service}}$ is only modified by transitions ${\tau^{mode_i mode_j}_{Service1}}|_{P_{1Service}} \in {T^{ModeSwitches}_{Service1}}|_{P_{1Service}}$. Furthermore, $t^{switch}_{Service1}|_{P_{1Service}}$ is assigned the value of *time* at which these transitions are taken. Therefore, based on the enabling conditions of ${\tau^{mode_i mode_j}_{Service1}}|_{P_{1Service}}$, for an arbitrary computation of $P_{1Service}$:

For $y \geq 0$

$$t^{switch}_{Service1}|_{P_{1Service}}(t) = f_a\big(ModeSwitchCheckTimes_{Service1}, ModeSwitchConditions_{Service1}(t-y)\big) \qquad (A1)$$

where

$ModeSwitchCheckTimes_{Service1} = \{$Set of time instants (relative to last mode switch time) at which mode switch conditions are checked according to the service description of CPS service $Service1\}$.

$ModeSwitchConditions_{Service1}(t) = \{$Set that contains the status at time t of all the mode switch assertions associated with CPS service $Service1\}$.

From the definition of $ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j)$, presented in Appendix B:

$$ModeSwithCheckTimes_{Service1} = f_b(ModePeriods_{Service1}, ModeSwitchFreqs_{Service1}) \qquad (B1)$$

Repeating the argument presented in the proof of Theorem 10.1,

$$ModeSwithConditions_{Service1}(t) = f_c(\sigma^{tr}_{P_{1Service}} \upharpoonright O_{in}) \qquad (C1)$$

Combining (A1), (B1), and (C1):

$$t^{switch}_{Service1}|_{P_{1Service}}(t) = f_d\big(ModePeriods_{Service1}, ModeSwitchFreqs_{Service1}$$
$$\sigma^{tr}_{P_{1Service}} \upharpoonright O_{in}\big) \qquad (D1)$$

From the transitions of $P_{kServices}$ (outlined in Appendix C), it can be seen that state variable $t^{switch}_{Service1}|_{P_{kServices}}$ is again only modified by transitions $\tau^{mode_i mode_j}_{Service1}|_{P_{kServices}}$ $\in T^{ModeSwitches}_{Service1}|_{P_{kServices}}$. Since enabling conditions and modification statements of $t^{switch}_{Service1}$ in $T^{ModeSwitches}_{Service1}|_{P_{kServices}}$ and $T^{ModeSwitches}_{Service1}|_{P_{1Service}}$ are identical, behavior of state variables $t^{switch}_{Service1}|_{P_{kServices}}$ and $t^{switch}_{Service1}|_{P_{1Service}}$ is identical. (Furthermore, based on Lemma I, a computation of $P_{kServices}$ cannot keep advancing time without taking the transitions associated with $Service1$.) Therefore, from (D1), in an arbitrary computation of $P_{kServices}$:

$$t^{switch}_{Service1}|_{P_{kServices}}(t) = f_d\big(ModePeriods_{Service1}, ModeSwitchFreqs_{Service1}$$
$$\sigma^{tr}_{P_{kServices}} \upharpoonright O_{in}\big) \qquad (E1)$$

From (D1) and (E1), it follows that

For arbitrary computations $\sigma_{P_{1Service}}$ *and* $\sigma_{P_{kServices}}$*, if*

$$\sigma^{tr}_{P_{1Service}} \textit{ under } O_{in} = \sigma^{tr}_{P_{kServices}} \textit{ under } O_{in}$$

then $\forall\ t \geq 0$

$$t^{switch}_{Service1}|_{P_{1Service}}(t) = t^{switch}_{Service1}|_{P_{kServices}}(t) \qquad \textbf{\textit{(Lemma II)}}$$

From the definition of t_{jump} used in the modification statements of state variable $t^{next}_{Service1}|_{P_{1Service}}$ in transitions $T^{Service1}_{P_{1Service}}$ (outlined in Appendix B), it can be seen that during an arbitrary computation of $P_{1Service}$:

$$t^{next}_{Service1}|_{P_{1Service}}(t) = f_e\big(t^{switch}_{Service1}|_{P_{1Service}}(t), ModePeriods_{Service1},$$
$$TaskFreqs_{Service1}\big) \qquad (A2)$$

From the transitions of $P_{kServices}$ (outlined in Appendix C), it can be seen that transitions $T^{Service1}_{P_{kServices}}$ and $T^{Service1}_{P_{1Service}}$ are identical in terms of modification statements of state variable $t^{next}_{Service1}|_{P_{kServices}}$ and $t^{next}_{Service1}|_{P_{1Service}}$ as well as the definition of t_{jump}

used in these modification statements. Therefore, from (A2), in an arbitrary computation of $P_{kServices}$:

$$t^{next}_{Service1}|_{P_{kServices}}(t) = f_e\big(t^{switch}_{Service1}|_{P_{kServices}}(t), ModePeriods_{Service1}, TaskFreqs_{Service1}\big) \qquad (B2)$$

From the combination of Lemma II, (A2), and (B2), it follows that

For arbitrary computations $\sigma_{P_{1Service}}$ *and* $\sigma_{P_{kServices}}$*, if*

$$\sigma^{tr}_{P_{1Service}} \textit{ under } O_{in} = \sigma^{tr}_{P_{kServices}} \textit{ under } O_{in}$$

then $\forall\ t \geq 0$

$$t^{next}_{Service1}|_{P_{1Service}}(t) = t^{next}_{Service1}|_{P_{kServices}}(t) \qquad \textbf{\textit{(Lemma III)}}$$

Let

$$TransitionSequence_{T^{Service1}_{P_{1Service}}} = \tau_0, \tau_1, \tau_2, \ldots$$

such that: 1) $\tau_i \in T^{Service1}_{P_{1Service}}$

2) In a computation of $P_{1Service}$, no transition $\tau_j \in T^{Service1}_{P_{1Service}}$ is taken after transition τ_i but before transition τ_{i+1}

$$TransitionSequence_{T^{Service1}_{P_{kServices}}} = \tau_0, \tau_1, \tau_2, \ldots$$

such that: 1) $\tau_i \in T^{Service1}_{P_{kServices}}$

2) In a computation of $P_{kServices}$, no transition $\tau_j \in T^{Service1}_{P_{kServices}}$ is taken after transition τ_i but before transition τ_{i+1}

From the description of $P_{1Service}$, presented in Appendix B, it can be seen that system starts in an initial state where $mode_{Service1} = mode_1$. Then, system keeps taking transition $\tau^{mode_1}_{Service1}$ until the time $t^{switch1}_{Service1}$ when system takes the transition $\tau^{mode_1 mode_j}_{Service1}$. ($mode_j$ depends on the status of mode switch assertions at time $t^{switch1}_{Service1}$.) Then, system keeps taking transition $\tau^{mode_j}_{Service1}$ until the time $t^{switch2}_{Service1}$ when system takes the transition $\tau^{mode_j mode_k}_{Service1}$ and so on. Based on this description, it can be argued that

$$TransitionSequence_{T^{Service1}_{P_{1Service}}} = f_h(ModeSwitchInstants^{Service1}_{P_{1Service}}, ModeSwitchConditionsSet^{Service1}_{P_{1Service}}) \qquad \textit{(A3)}$$

where

$ModeSwitchInstants_{P_{1Service}}^{Service1} = \{$Set of all values that are assigned to state variable $t_{Service1}^{switch1}|_{P_{1Service}}$ during a computation of $P_{1Service}\}$

$= \{t_{Service1}^{switch1}|_{P_{1Service}}, t_{Service1}^{switch2}|_{P_{1Service}}, \ldots, \}$

$ModeSwitchConditionsSet_{P_{1Service}}^{Service1} =$

$\{ModeSwitchConditions_{Service1}(t_{Service1}^{switch1}|_{P_{1Service}}),$

$ModeSwitchConditions_{Service1}(t_{Service1}^{switch2}|_{P_{1Service}}), \ldots\}$

From the description of $P_{kServices}$, presented in Appendix C, it can be seen that system starts in an initial state where $mode_{Service1} = mode_1$. Then, from the set of transitions $T_{P_{kServices}}^{Service1}$, system keeps on taking only the transition $\tau_{Service1}^{mode_1}$ until the time $t_{Service1}^{switch1}$ when system takes the transition $\tau_{Service1}^{mode_1 mode_j}$ from $T_{P_{kServices}}^{Service1}$. ($mode_j$ depends on the status of mode switch assertions of $Service1$ at time $t_{Service1}^{switch1}$.) Then, from the set of transitions $T_{P_{kServices}}^{Service1}$, system again keeps taking only the transition $\tau_{Service1}^{mode_j}$ until the time $t_{Service1}^{switch2}$ when system takes the transition $\tau_{Service1}^{mode_j mode_k}$ from $T_{P_{kServices}}^{Service1}$ and so on. Based on this description, it can be argued that

$$TransitionSequence_{T_{P_{kServices}}^{Service1}} = f_h(ModeSwitchInstants_{P_{kServices}}^{Service1}, ModeSwitchConditionsSet_{P_{kServices}}^{Service1}) \qquad (B3)$$

where

$ModeSwitchInstants_{P_{kServices}}^{Service1} = \{$Set of all values that are assigned to state variable $t_{Service1}^{switch1}|_{P_{kServices}}$ during a computation of $P_{kServices}\}$

$= \{t_{Service1}^{switch1}|_{P_{kServices}}, t_{Service1}^{switch2}|_{P_{kServices}}, \ldots, \}$

$ModeSwitchConditionsSet_{P_{kServices}}^{Service1} =$

$\{ModeSwitchConditions_{Service1}(t_{Service1}^{switch1}|_{P_{kServices}}),$

$ModeSwitchConditions_{Service1}(t_{Service1}^{switch2}|_{P_{kServices}}), \ldots\}$

if $\sigma_{P_{1Service}}^{tr}$ under $O_{in} = \sigma_{P_{kServices}}^{tr}$ under O_{in}, then from Lemma II,

$$ModeSwitchInstants_{P_{kServices}}^{Service1} = ModeSwitchInstants_{P_{1Service}}^{Service1} \qquad (C3)$$

Moreover, given that $\sigma_{P_{1Service}}^{tr}$ under $O_{in} = \sigma_{P_{kServices}}^{tr}$ under O_{in}, from the combination of (C1) and Lemma II:

$$ModeSwitchConditionsSet_{P_{kServices}}^{Service1} = \\ ModeSwitchConditionsSet_{P_{1Service}}^{Service1} \qquad \textit{(D3)}$$

Therefore, by combination of (A3), (B3), (C3), and (D3), it follows that:

For arbitrary computations $\sigma_{P_{1Service}}$ and $\sigma_{P_{kServices}}$, if

$$\sigma_{P_{1Service}}^{tr} \text{ under } O_{in} = \sigma_{P_{kServices}}^{tr} \text{ under } O_{in}$$

then

$$TransitionSequence_{T_{P_{kServices}}^{Service1}} = \\ TransitionSequence_{T_{P_{1Service}}^{Service1}} | f_{MapService1} \qquad \textbf{\textit{(Lemma IV)}}$$

where

$TransitionSequence_{T_{P_{1Service}}^{Service1}} | f_{MapService1} =$ A $TransitionSequence$ obtained by replacing each transition τ in $TransitionSequence_{T_{P_{1Service}}^{Service1}}$ with $f_{MapService1}(\tau)$

Equipped with Lemmas I-IV, presented above, the proof of Theorem 10.2 can now be pursued as follows:

As outlined in Section 10.3, the temporally reduced behavior of a transition system P under observable variables O ($\sigma_P^{tr} \upharpoonright O$) can also be represented as follows:

$$\sigma_P^{tr} \upharpoonright O = f(\Delta_{T_P^{relevant} \upharpoonright O}, TimedTransitionSequence_{T_P^{relevant} \upharpoonright O}) \qquad (A)$$

where

$T_P^{relevant} \upharpoonright O = \{\tau_i \mid (\tau_i \in T_P) \wedge$ (modification statements of transition τ_i change the value of non-time state variables in O)$\}$

$\Delta_{T_P^{relevant} \upharpoonright O} = \{\Delta_{\tau_i} \mid \tau_i \in T_P^{relevant} \upharpoonright O\}$

$\Delta_{\tau_i} = \{\Delta s_{\tau_i} \mid s$ is a non-time state variable in $O\}$

$\Delta s_{\tau_i} = s'_{\tau_i} - s_{\tau_i} =$ Change in the value of state variable s, caused by transition τ_i

$TimedTransitionSequence_{T_P^{relevant} \upharpoonright O} = (t_0, \tau_0), (t_1, \tau_1), (t_2, \tau_2), \ldots$

such that:

1) for each element $(t_i, \tau_i), \tau_i \in T_P^{relevant} \upharpoonright O$ and

system reaches *time* t_i after transition τ_i is taken.

2) $t_{i+1} \geq t_i$

Specializing (A) for transition system $P_{1Service}$ and observable variables O_{out}:

$$\sigma^{tr}_{P_{1Service}} \upharpoonright O_{out} = f_1(\Delta_{T^{relevant}_{P_{1Service}} \upharpoonright O_{out}}, \\ TimedTransitionSequence_{T^{relevant}_{P_{1Service}} \upharpoonright O_{out}}) \qquad (B)$$

Specializing (A) for transition system $P_{kServices}$ and observable variables O_{out}:

$$\sigma^{tr}_{P_{kServices}} \upharpoonright O_{out} = f_1(\Delta_{T^{relevant}_{P_{kServices}} \upharpoonright O_{out}}, \\ TimedTransitionSequence_{T^{relevant}_{P_{kServices}} \upharpoonright O_{out}}) \qquad (C)$$

From the Manna-Pnueli Transition System Representation $P_{1Service}$, outlined in Appendix B, it can be seen that

$$T^{relevant}_{P_{1Service}} \upharpoonright O_{out} = T^{ModeSwitches}_{Service1}|_{P_{1Service}} \cup T^{TimeIncrement}_{Service1}|_{P_{1Service}} \qquad (D)$$

where

$T^{ModeSwitches}_{Service1}|_{P_{1Service}}$ = Set of transitions $T^{ModeSwitches}_{Service1}$, as defined in $P_{1Service}$

$T^{TimeIncrement}_{Service1}|_{P_{1Service}}$ = Set of transitions $T^{TimeIncrement}_{Service1}$, as defined in $P_{1Service}$

From the Manna-Pnueli Transition System Representation $P_{kServices}$, outlined in Appendix C, it can be seen that only the transitions associated with CPS service $Service1$ modify the observable state variables in O_{out}. Therefore,

$$T^{relevant}_{P_{kServices}} \upharpoonright O_{out} = T^{ModeSwitches}_{Service1}|_{P_{kServices}} \cup T^{TimeIncrement}_{Service1}|_{P_{kServices}} \qquad (E)$$

where

$T^{ModeSwitches}_{Service1}|_{P_{kServices}}$ = Set of transitions $T^{ModeSwitches}_{Service1}$, as defined in $P_{kServices}$

$T^{TimeIncrement}_{Service1}|_{P_{kServices}}$ = Set of transitions $T^{TimeIncrement}_{Service1}$, as defined in $P_{kServices}$

Since definitions of transitions $T^{ModeSwitches}_{Service1}$ and $T^{TimeIncrement}_{Service1}$ in both $P_{1Service}$ (Appendix B) and $P_{kServices}$ (Appendix C) have exaclty the same modification statements for observable variables O_{out}, the following can be inferred from (D) and (E):

$$\Delta_{T^{relevant}_{P_{1Service}} \upharpoonright O_{out}} = \Delta_{T^{relevant}_{P_{kServices}} \upharpoonright O_{out}} \qquad (F)$$

Combining Lemma III, Lemma IV, and definition of $TimedTransitionSequence$, it follows that:

For arbitrary computations $\sigma_{P_{1Service}}$ *and* $\sigma_{P_{kServices}}$, *if*

$\sigma^{tr}_{P_{1Service}}$ under $O_{in} = \sigma^{tr}_{P_{kServices}}$ under O_{in}

then

$$TimedTransitionSequence_{T^{Service1}_{P_{kServices}}} = TimedTransitionSequence_{T^{Service1}_{P_{1Service}}} | f_{MapService1} \qquad \text{(G)}$$

where

$TimedTransitionSequence_{T^{Service1}_{P_{1Service}}} | f_{MapService1} =$ A $TimedTransitionSequence$ obtained by replacing each transition τ in $TimedTransitionSequence_{T^{Service1}_{P_{1Service}}}$ with $f_{MapService1}(\tau)$

Combining (G) with information about relevant transitions in (D) and (E), it follows that

For arbitrary computations $\sigma_{P_{1Service}}$ *and* $\sigma_{P_{kServices}}$, *if*

$\sigma^{tr}_{P_{1Service}}$ under $O_{in} = \sigma^{tr}_{P_{kServices}}$ under O_{in}

then

$$TimedTransitionSequence_{T^{relevant}_{P_{kServices}}} \restriction O_{out} = TimedTransitionSequence_{T^{relevant}_{P_{1Service}}} \restriction O_{out} \qquad \text{(H)}$$

Combining (B), (C), (F), and (H), it follows:

For arbitrary computations $\sigma_{P_{1Service}}$ *and* $\sigma_{P_{kServices}}$, *if*

$\sigma^{tr}_{P_{1Service}} \restriction O_{in} = \sigma^{tr}_{P_{kServices}} \restriction O_{in}$

then

$\sigma^{tr}_{P_{1Service}} \restriction O_{out} = \sigma^{tr}_{P_{kServices}} \restriction O_{out}$

$\square$

CHAPTER XI

SIMULATION-BASED SMART GRID TESTBEDS: DEMONSTRATING THE ADVANTAGES OF SERVICE-ORIENTED CPS REFERENCE MODEL

This dissertation has presented a service-oriented CPS reference model and associated technologies that can address the unique challenges posed by the emerging CPS application areas that are characterized by their larger scale and "always online" nature. Smart grid [75] provides a prime example of the above mentioned large scale and "always online" CPS application domain. Due to the safety-critical nature of the smart grid infrastructure, simulation-based smart grid testbeds play a central role for research efforts in this area. This chapter presents simulation-based smart grid testbeds that can be used to demonstrate the advantages of applying the proposed service-oriented CPS approach (as compared to the traditional task-based computing model or enterprise-domain service-oriented computing model) to smart grid applications in a virtual environment before future steps are taken towards the implementation of this service-oriented CPS approach on live smart grid infrastructure.

11.1 Smart Grid Testbed: Traditional Service-Oriented Computing

This section presents the design of a simulation-based smart grid testbed that assumes the application of enterprise-domain service-oriented computing technologies (Web Services) for implementing smart grid applications. As shown in Figure 11.1, this smart grid testbed combines a state-of-the-art network simulator, ns-3 [57], and

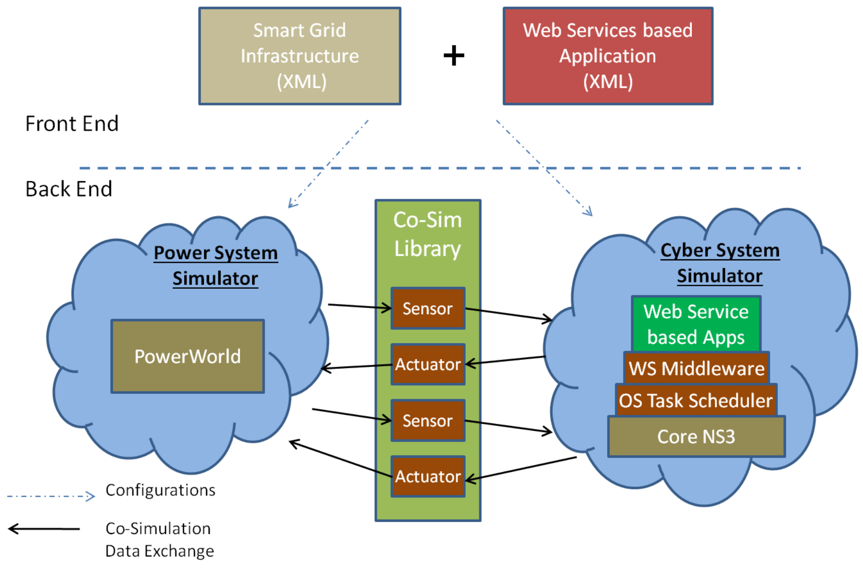


Figure 11.1: Structure of the simulation-based smart grid testbed with traditional enterprise-domain, service-oriented computing paradigm.

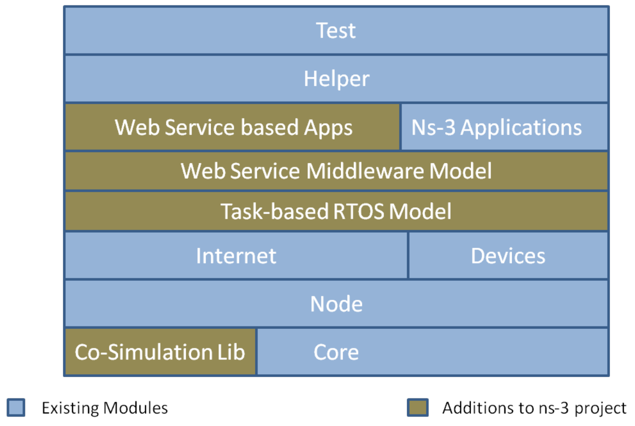


Figure 11.2: Additions to the standard structure of ns-3 network simulator as a component of smart grid testbed with traditional SOC paradigm.

a state-of-the-art power system simulator, PowerWorld [60]. The proposed simulation environment also extends ns-3 with a cyber-physical co-simulation library [73], model of an operating system's task scheduler, model of inter-node message transport using Web Services middleware, and Web Services based smart grid applications. Figure 11.2 shows the overall organization of ns-3 software after the additions that have been made to ns-3 as a part of this simulation-based smart grid tested environment. As shown later in this chapter, this smart grid testbed environment can be used to explore the pitfalls of applying the enterprise-domain service-oriented computing technologies (Web Services) for implementing smart grid applications in a virtual environment.

11.2 Smart Grid Testbed: Proposed Service-Oriented CPS Approach

This section presents the design of a simulation-based smart grid testbed that assumes the application of service-oriented CPS reference model and associated technologies (proposed in this dissertation) for implementing smart grid applications. As shown in Figure 11.3, this smart grid testbed [73] combines a state-of-the-art network simulator, ns-3 [57], and a state-of-the-art power system simulator, PowerWorld [60]. The proposed simulation environment also extends ns-3 with a cyber-physical co-simulation library [73], model of the proposed Giotto-based service deployment platform, and CPS Services based smart grid applications. Figure 11.4 shows the overall organization of ns-3 software after the additions that have been made to ns-3 as a part of this simulation-based smart grid tested environment.

Since the smart grid infrastructure is a safety-critical system, any new ideas about its operation must first be demonstrated in a virtual environment. Therefore, this simulation-based smart grid testbed can be extremely useful in demonstrating the application of the service-oriented CPS reference model and associated technologies, proposed in this dissertation, to existing as well as future smart grid applications in

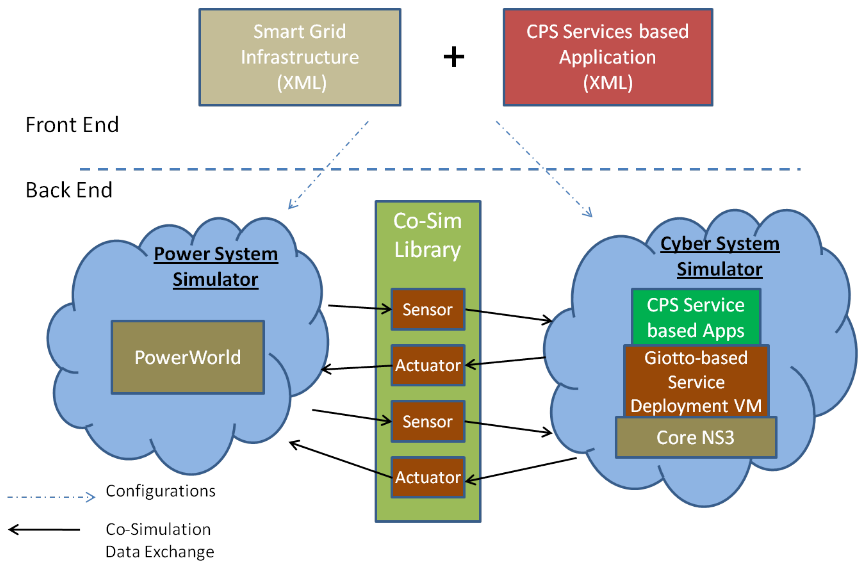


Figure 11.3: Structure of the simulation-based smart grid testbed with proposed CPS-enabled, service-oriented computing paradigm.

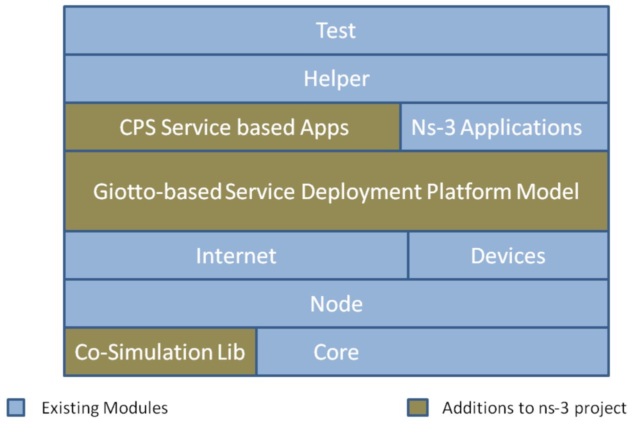


Figure 11.4: Additions to the standard structure of ns-3 network simulator as a component of smart grid testbed with the proposed CPS-enabled SOC paradigm.

a virtual environment before further steps are taken towards the implementation of this service-oriented CPS approach on live smart grid infrastructure.

11.3 Smart Grid Case Study: Demonstration of the Advantages of Proposed Service-Oriented CPS Approach

This section uses the smart grid case study of Chapter 5 to compare the performance of three implementation options: 1) task-based embedded control systems approach, 2) traditional enterprise-domain, service-oriented computing approach, and 3) service-oriented CPS approach (proposed in this dissertation).

11.3.1 Smart Grid Case Study: Task-based Embedded Control Systems Approach

Smart grid case study of Chapter 5 implements the power agreement application on the same real-time computing platform after it has successfully supported the operation of a demand response application for a period of time. According to the task-based embedded control systems approach, used in the domain of automotive and avionics, if the same real-time computing platform is to be used for implementing another feedback controller at any time after the initial system development, the system must be taken out of operation so that the task-based real-time control code could be changed and tested. However, taking the smart grid infrastructure out of operation for installing a new application is not practical. Therefore, task-based approach used in the automotive and avionics domain cannot be used for smart grid domain, because this approach cannot support the "always online" nature of smart grid infrastructure.

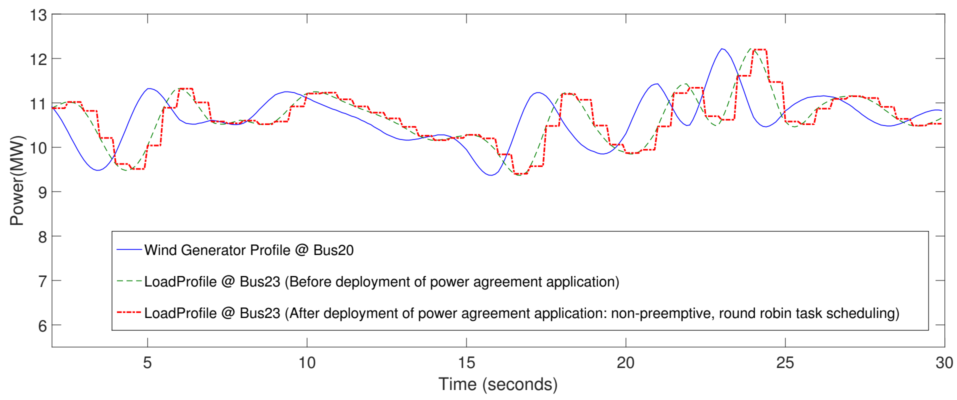


Figure 11.5: Demand response application from case study in Chapter 5: performance comparison before and after the deployment of the power agreement application (resource overloading on ProsumerCompNode computing platforms for prousmer1 and prosumer10).

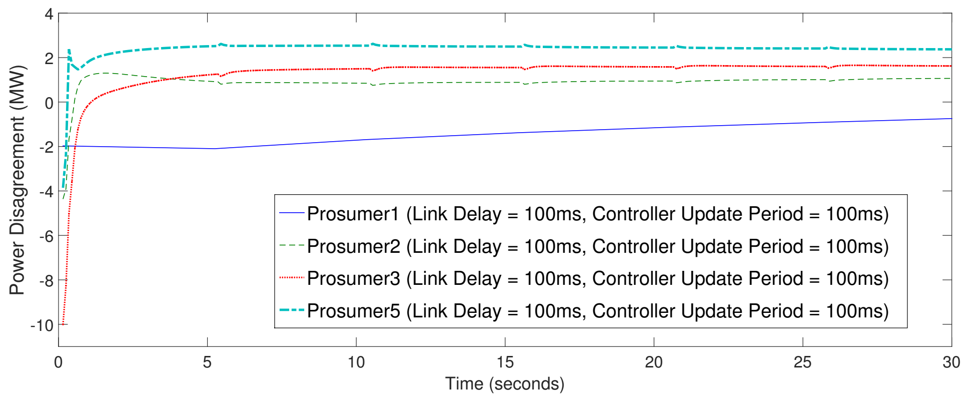


Figure 11.6: Power agreement application from case study in Chapter 5: convergence behavior under resource overloading on ProsumerCompNode computing platforms for prousmer1 and prosumer10 (only 4 out of 10 prosumers are depicted for readability).

11.3.2 Smart Grid Case Study: Traditional Enterprise-Domain Service-Oriented Computing Approach

Through the service description, service publication to a service repository, and service discovery mechanisms of traditional enterprise-domain, service-oriented computing paradigm, it is possible to deploy power agreement application (of case study in Chapter 5) on the smart grid infrastructure without taking the system out of operation. However, traditional Web Services based SOC technologies do not support "resource-aware" service deployment. As a result, the deployment of new services associated with power agreement application might result in resource overloading of the underlying computing platform, adversely affecting the timing performance of old as well as the new services.

Using the case study of Chapter 5 and the smart grid testbed of Section 11.1, Figure 11.5 compares the performance of demand response application before and after the deployment of power agreement application, when the deployment of new services associated with power agreement application overloads some computing nodes. Furthermore, in this scenario, the performance (convergence behavior) of newly deployed power agreement application is also not satisfactory as shown in Figure 11.6, because the power agreement application fails to converge in the allotted 30 seconds.

11.3.3 Smart Grid Case Study: Proposed Service-Oriented CPS Approach

Through the CPS Service Description Languages (presented in Chapter 8) and CPS service deployment platform (presented in Chapter 9), the service-oriented CPS approach, proposed in this dissertation, can support "resource-aware" deployment of a CPS service on a computing node in the field. As a result, power agreement application (of case study in Chapter 5) can be deployed on the smart grid infrastructure without taking the system out of operation and any resource overloading conditions are detected at the deployment time. Therefore, the proposed CPS approach allows

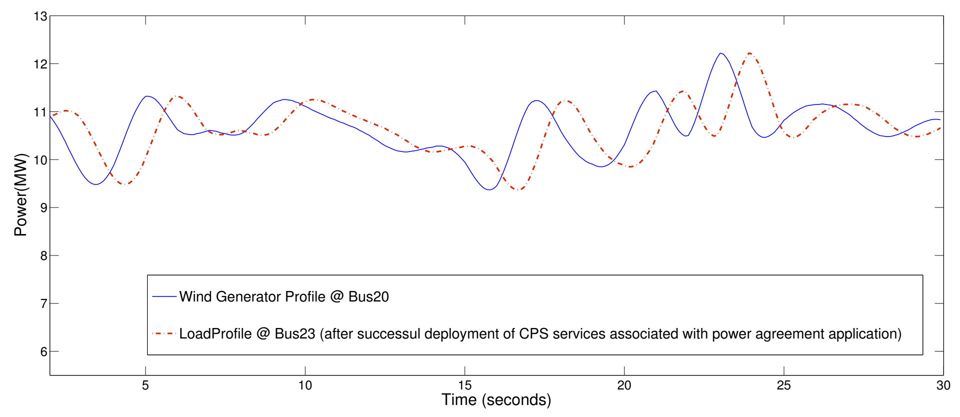


Figure 11.7: Demand response application from case study in Chapter 5: performance after the successful deployment of CPS services associated with power agreement application.

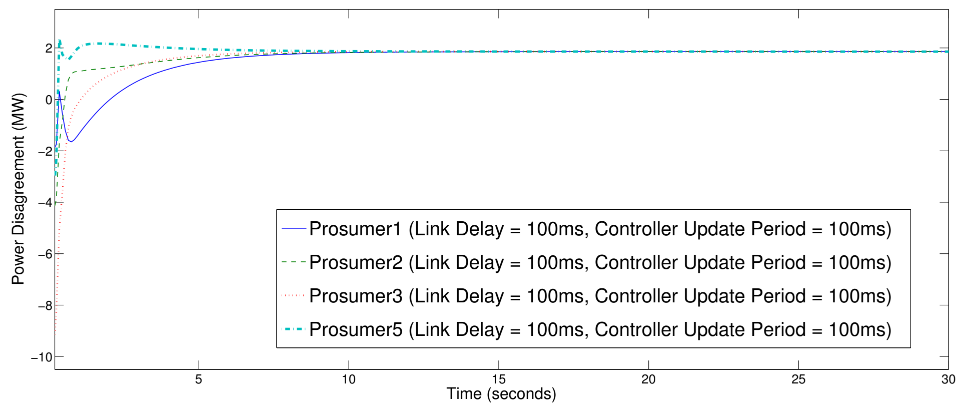


Figure 11.8: Power agreement application from case study in Chapter 5: convergence behavior after successful field deployment through CPS services on a computing infrastructure that was already supporting a demand response application (only 4 out of 10 prosumers are depicted for readability).

system reconfiguration while avoiding any surprise runtime timing constraint failures that might create unsafe conditions for the smart grid system. Using the smart grid testbed of Section 11.2, Figure 11.7 and Figure 11.8 show the results from implementing the case study of Chapter 5 through the proposed service-oriented CPS approach.

CHAPTER XII

CONCLUSION

Availability of cost-effective communication and computation technologies has enabled the development of a new breed of embedded control systems that are characterized by their larger scale, longer life-cycles, and "always-online" nature. Some prime examples of such systems are smart grid, vehicular networks, and automated irrigation networks. The development of this new breed of systems through traditional embedded control system development techniques (employed in the fields of automotive and avionics) will result in prohibitively high development and maintenance costs, because these traditional techniques are unable to support disruption-free incremental system deployment and reconfiguration that are fundamental requirements for handling the larger scale and "always-online" nature of this new breed of systems.

Emerging research area of cyber-physical sytems (CPS) aims to address the limitations of traditional embedded control system techniques by developing an integrated theory as well as an integrated development toolset for controller design and controller implementation phases of embedded control system development process. Although CPS research has resulted in a set of isolated theoretical results and development technologies, it lacks a holistic framework that can enable the development of a consistent set of theoretical results and development toolset for the emerging CPS application domains of smart grid and vehicular networks, characterized by their larger scale and "always-online" nature. In the past, various engineering domains have successfully employed the concept of a "reference model" to enable clear communication among stakeholders and to serve as the underlying framework for development of a consistent set of standards and technologies for that domain.

12.1 Summary of Contributions

This dissertation has formalized a service-oriented computing (SOC) based approach to cyber-physical systems (CPS) in the form of a *service-oriented CPS reference model*. The proposed reference model extends the traditional SOC paradigm for handling hard real-time aspects of the domain of cyber-physical systems by introducing resource-aware service deployment and quality-of-service (QoS)-aware service operation phases with certain formal performance guarantees. The proposed reference model also requires the existence of formal guarantees for the following aspects: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and (2) non-interference between the co-deployed CPS services from the perspective of their timing performance. The existence of these formal guarantees will provide a provably-correct process for converting a new CPS application from a CPS design specification to a service-based CPS deployment in the field without affecting the performance of already deployed CPS applications. As a result, unlike the task-based reference model from the domains of automotive and avionics, the proposed service-oriented CPS reference model will enable disruption-free incremental system deployment and reconfiguration that are fundamental requirements of the emerging safety-critical but large scale and "always-online" CPS application domains such as smart grid and vehicular networks.

Although the development of suitable technologies for a domain according to the requirements of a reference model for that domain is intended to be an on-going effort by a research community, this dissertation has made significant contributions to this effort by proposing solutions for the following technological requirements of service-oriented CPS reference model:

- CPS design specification language.
- simulation environment for CPS design refinement.

- service description language.
- service-based computing platform for CPS computing nodes with support for resource-aware service deployment and QoS-aware service interaction.

By extending and applying the Manna-Pnueli Approach of formal methods for reactive computer systems, this dissertation has also shown how the aforementioned technological solutions combine to provide the formal performance guarantees, mandated by the proposed reference model. Finally, this dissertation has also presented simulation-based smart grid testbeds that can be used to demonstrate the advantages of the proposed service-oriented CPS approach in a virtual environment before its implementation on safety-critical, live smart grid infrastructure.

12.2 Future Directions

This dissertation has presented a set of solutions for the technological requirements of the proposed service-oriented CPS reference model that are based on Giotto programming language. Giotto is a research-grade programming language that has been demonstrated on the embedded computing platform for robotics and avionics [26]. For transitioning the proposed technologies to live smart grid infrastructure, development of Giotto compilers for embedded computing platforms used in the domain of power systems will be an important step.

It must be emphasized that the concept of a reference model and associated technological requirements allows a research community to investigate and compare multiple solution approaches for meeting these technological requirements [58] [71]. Therefore, in future, there could be multiple candidate solutions for meeting each of the technological requirements of the service-oriented CPS reference model, proposed in this dissertation. However, in order to achieve the goals of disruption-free evolution and reconfiguration of safety critical but large scale and "always-online" CPS application domains (such as smart grid), any candidate set of solutions must ensure the existence of formal guarantees for the following aspects: (1) functional equivalence between a CPS design specification and the corresponding service-based CPS field deployment and (2) non-interference between the co-deployed services from the perspective of their timing performance.

APPENDIX A

MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE IN CPS-DSL

A *ComputingNode* block, $CompNode1$, of CPS-DSL can be represented as the following Manna-Pnueli Transition System, $P_{CompNode} < \Pi_{P_{CompNode}}, \Sigma_{P_{CompNode}}, T_{P_{CompNode}}, \Theta_{P_{CompNode}} >$:

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

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

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 $CompNode1$,

- $\mathbf{\Sigma_{P_{CompNode}}}$ — 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 $\mathbb{R}_{\geq 0}$. 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 $\mathbb{D}$, all the state variables in Π_α have the domain $\mathbb{D}$:

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

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

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

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

- $\mathbf{T_{P_{CompNode}}}$ — 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 Chapter 10, each transition τ can be characterized by an *enabling condition* and a *set of modification statements*. Based on the above mentioned set of transitions $T_{P_{CompNode}}$ of $P_{CompNode}$, all the diligent transitions of $P_{CompNode}$ 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*

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

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.

$ModeSwitchCheckTime_{CompNode1}(t, t^{switch}_{Service1}, mode_i, mode_j)$ = 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. $outMsgPorts_{CompNode1}^{mode_j}{}' =$

   $ControllerOutsToOutMsgs_{CompNode1}^{mode_j}$

   $(periodicControllerOuts_{CompNode1}^{mode_j}{}')$

   where

   $ControllerOutsToOutMsgs_{CompNode1}^{mode_j}$ = 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 $CompNode1$.

7. $periodicControllerIns_{controllerFucntion_b}{}' =$

   $LoadControllerInputs_{controllerFunction_b}^{mode_j}(sensePorts_{CompNode1}^{mode_j}{}',$

   $inMsgPorts_{CompNode1}^{mode_j}{}', periodicControllerOuts_{CompNode1}^{mode_j}{}')$

   for every $controllerFunction_b \in ControllerFunctions_{CompNode1}^{mode_j}$

   where

   $LoadControllerInputs_{controllerFunction_b}^{mode_j}$ = 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) $\tau_{CompNode1}^{mode_i}$*: Enabling Condition*

$$C_{\tau_{CompNode1}^{mode_i}} = (mode_{CompNode1} == mode_i)$$

$$\wedge \neg(ModeSwitchCondition_{CompNode1}(t, mode_i, mode_c) \wedge$$

$$ModeSwitchCheckTime_{CompNode1}(t, t^{switch}_{CompNode1}, mode_i, mode_c))$$

$\forall$ $mode_c \in \{mode_c \mid \exists$ a *mode switch* from $mode_i$ to $mode_c$ 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

   $$t_{jump} = \min\Big\{t_j \mid (t_j > 0) \wedge (t' + t_j = t^{switch}_{Service1} + a\{\frac{Period_{mode_i}}{ControllerFucntionFreq_{controllerFunction_d}}\})$$

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

   and

   for some $controllerFunction_d \in ControllerFunctions^{mode_i}_{CompNode1}$ $\Big\}$

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

   $\forall$ $controllerFunction_e \in \Big\{controllerFunction_e \mid$
   $(controllerFunction_e \in ControllerFunctions^{mode_i}_{CompNode1})$
   $\wedge$ $(t' = t^{switch}_{CompNode1} + a\{\frac{Period_{mode_i}}{ControllerFunctionFreq_{controllerFunction_e}}\})$
   for some $a \in \{1, 2, \ldots, ControllerFunctionFreq_{controllerFunction_e}\}\Big\}$

   where

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

4. $periodicControllerIns_{controllerFunction_f}{}' =$

$$LoadControllerInputs^{mode_i}_{controllerFunction_f}(sensePorts^{mode_i}_{CompNode1}{}',$$

$$inMsgPorts^{mode_i}_{CompNode1}{}', periodicControllerOuts^{mode_i}_{CompNode1}{}')$$

$\forall\ controllerFunction_f \in \Big\{ controllerFunction_f \mid$

$(controllerFunction_f \in ControllerFunctions^{mode_i}_{CompNode1})$

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

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

5. $actPorts^{mode_i}_{CompNode1}{}' =$

$ControllerOutsToActs^{mode_i}_{CompNode1}(periodicControllerOuts^{mode_i}_{CompNode1}{}')$

6. $outMsgPorts^{mode_i}_{CompNode1}{}' =$

$ControllerOutsToOutMsgs^{mode_i}_{CompNode1}$

$$(periodicControllerOuts^{mode_i}_{CompNode1}{}')$$

- $\mathbf{\Theta_{P_{CompNode}}}$ — An *initial condition*.

Any initial state s of transition system $P_{CompNode}$ must satisfy the following initial conditions:

$t = 0$

$t^{switch}_{CompNode1} = 0$

$mode_{CompNode1} = mode_1$

$t^{next}_{CompNode1} = \min \Big\{ t_j \mid (t_j > 0) \wedge (t_j = a\{\frac{Period_{mode_1}}{ControllerFunctionFreq_{controllerFunction_d}}\})$

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

for some $controllerFunction_d \in ControllerFunctions^{mode_1}_{CompNode1} \Big\}$

APPENDIX B

MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE WITH 1 CPS SERVICE

A CPS computing node with one successfully deployed Giotto-based CPS service, $Service1$, can be represented as the following Manna-Pnueli Transition System, $P_{1Service} < \Pi_{P_{1Service}}, \Sigma_{P_{1Service}}, T_{P_{1Service}}, \Theta_{P_{1Service}} >$:

- $\mathbf{\Pi_{P_{1Service}}}$ — A finite set of *state variables.*

  $$\begin{aligned}\Pi_{P_{1Service}} = \{&t, t^{switch}_{Service1}, mode_{Service1}, t^{next}_{Service1},\\ &sensePort^1_{Service1}, sensePort^2_{Service1}, \ldots, sensePort^p_{Service1},\\ &inMsgPort^1_{Service1}, inMsgPort^2_{Service1}, \ldots, inMsgPort^r_{Service1},\\ &actPort^1_{Service1}, actPort^2_{Service1}, \ldots, actPort^q_{Service1},\\ &outMsgPort^1_{Service1}, outMsgPort^2_{Service1}, \ldots, outMsgPort^l_{Service1},\\ &taskInPort^1_{Service1}, taskInPort^2_{Service1}, \ldots, taskInPort^a_{Service1},\\ &taskOutPort^1_{Service1}, taskOutPort^2_{Service1}, \ldots, taskOutPort^b_{Service1},\\ &taskPvtPort^1_{Service1}, taskPvtPort^2_{Service1}, \ldots, taskPvtPort^c_{Service1}\}\end{aligned}$$

  where

  t = time,

  $t^{switch}_{Service1}$ = latest mode switch time of CPS service, $Service1$,

  $mode_{Service1}$ = current mode of CPS service, $Service1$,

  $t^{next}_{Service1}$ = next relevant time instant (task update, actuator update, output message update) during the operation of CPS service $Service1$ in its current mode,

$sensePort^i_{Service1}$ = sensor port of CPS service $Service1$,

$inMsgPort^i_{Service1}$ = input message port of CPS service $Service1$,

$actPort^i_{Service1}$ = actuator port of CPS service $Service1$,

$outMsgPort^i_{Service1}$ = output message port of CPS service $Service1$,

$taksInPort^i_{Service1}$ = input port of a task in CPS service $Service1$,

$taskOutPort^i_{Service1}$ = output port of a task in CPS service $Service1$,

$taskPvtPort^i_{Service1}$ = private port of a task in CPS service $Service1$,

- $\mathbf{\Sigma_{P_{1Service}}}$ — 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, $t^{switch}_{Service1}$, and $t^{next}_{Service1}$ is $\mathbb{R}_{\geq 0}$. The domain of state variable $mode_{Service1}$ is $\mathbb{M}_{Service1}$ = {Set of modes of CPS service $Service1$}. Given the following definitions of Π_α and $\mathbb{D}$, all the state variables in Π_α have the domain $\mathbb{D}$:

$$\begin{aligned}\Pi_\alpha = \{&sensePort^i_{Service1}, actPort^i_{Service1}, outMsgPort^i_{Service1},\\ &taskInPort^i_{Service1}, taskOutPort^i_{Service1}, taskPvtPort^i_{Service1}\}\end{aligned}$$

$$\begin{aligned}\mathbb{D} = \{x \mid\ &(x \in \mathbb{R})\\ &\wedge (x \text{ can be represented by type } \textit{double} \text{ of computer system})\}\end{aligned}$$

  The state variable $inMsgPort^i_{Service1}$ has the following domain:

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

- $\mathbf{T_{P_{1Service}}}$ — A finite set of transitions.

$$T_{P_{1Service}} = \tau_I \cup T^{ModeSwitches}_{Service1} \cup T^{TimeIncrement}_{Service1}$$

where $\quad \tau_I =$ Idling Transition

$$T^{ModeSwitches}_{Service1} = \{\tau^{mode_i mode_j}_{Service1} \mid \exists \text{ a } \textit{mode switch} \text{ from } mode_i \text{ to } mode_j \text{ in CPS service } Service1\}$$

$$T^{TimeIncrement}_{Service1} = \{\tau^{mode_1}_{Service1}, \tau^{mode_2}_{Service1}, \ldots, \tau^{mode_M}_{Service1}\}$$

As outlined in the summary of Manna-Pnueli Transition System approach (Chapter 10), each transition τ can be characterized by an *enabling condition* and a *set of modification statements.* Based on the above mentioned set of transitions $T_{P_{1Service}}$ of $P_{1Service}$, all the diligent transitions of $P_{1Service}$ can be completely described through the enabling conditions and modification statements of the following generic transitions: $\tau^{mode_i mode_j}_{Service1}$ and $\tau^{mode_i}_{Service1}$.

a) $\tau^{mode_i mode_j}_{Service1}$*: Enabling Condition*

$$\begin{aligned} C_{\tau^{mode_i mode_j}_{Service1}} = {} & (mode_{Service1} == mode_i) \\ & \wedge\ ModeSwitchCondition_{Service1}(t, mode_i, mode_j) \\ & \wedge\ ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j) \end{aligned}$$

where

$ModeSwitchCondition_{Service1}(t, mode_i, mode_j) =$ An assertion that returns true if the *guard condition* associated with the *driver* of *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service1$ is true at time t.

$ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j) =$ An assertion that returns true if $t - t^{switch}_{Service1} = a\{\frac{Period_{mode_i}}{SwitchFreq_{mode_i mode_j}}\}$, for some $a \in \{1, 2, \ldots, SwitchFreq_{mode_i mode_j}\}$.

b) $\tau^{mode_i mode_j}_{Service1}$*: Modification Statements*

1. $mode_{Service1}{}' = mode_j$

2. $t^{switch}_{Service1}{}' = t$

3. $t^{next}_{Service1}{}' = t + t_{jump}$

where

$$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t + t_j = t^{switch}_{Service1}{}' + a\{\frac{Period_{mode_j}}{TaskFreq_{task_d}}\}$$

for some $a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$

and for some $task_d \in Tasks^{mode_j}_{Service1}) \}$

4. $taskOutPorts^{mode_j}_{Service1}{}' =$

$$ModeSwitchFunction^{mode_i mode_j}_{Service1}(taskOutPorts^{mode_i}_{Service1})$$

where

$ModeSwitchFunction^{mode_i mode_j}_{Service1}$ = The function used in the definition of the *driver* associated with the *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service1$

5. $actPorts^{mode_j}_{Service1}{}' = TaskOutsToActs^{mode_j}_{Service1}(taskOutPorts^{mode_j}_{Service1}{}')$

where

$TaskOutsToActs^{mode_j}_{Service1}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the actuator ports in $mode_j$ of CPS service $Service1$.

6. $outMsgPorts^{mode_j}_{Service1}{}' =$

$$TaskOutsToOutMsgs^{mode_j}_{Service1}(taskOutPorts^{mode_j}_{Service1}{}')$$

where

$TaskOutsToOutMsgs^{mode_j}_{Service1}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the output message ports in $mode_j$ of CPS service $Service1$.

7. $$taskInPorts_{task_b}{}' = LoadTaskInputs_{task_b}^{mode_j}(sensePorts_{Service1}^{mode_j}{}', inMsgPorts_{Service1}^{mode_j}{}', taskOutPorts_{task_b}{}')$$

for every $task_b \in Tasks_{Service1}^{mode_j}$

where

$LoadTaskInputs_{task_b}^{mode_j}$ = A function that captures input-output relationship (produced by the combined effect) of all the *drivers*, updating the task input ports of $task_b$ in $mode_j$ of CPS service $Service1$.

c) $\tau_{Service1}^{mode_i}$: Enabling Condition

$$C_{\tau_{Service1}^{mode_i}} = (mode_{Service1} == mode_i) \wedge \neg(ModeSwitchCondition_{Service1}(t, mode_i, mode_c) \wedge ModeSwitchCheckTime_{Service1}(t, t_{Service1}^{switch}, mode_i, mode_c))$$

$\forall\ mode_c \in \{mode_c \mid \exists$ a *mode switch* from $mode_i$ to $mode_c$ of CPS service $Service1$ $\}$

d) $\tau_{Service1}^{mode_i}$: Modification Statements

1. $t' = t_{Service1}^{next}$

2. $t_{Service1}^{next}{}' = t' + t_{jump}$

where

$$t_{jump} = \min\left\{t_j \mid (t_j > 0) \wedge (t' + t_j = t_{Service1}^{switch} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_d}}\}) \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_d}\} \text{ and for some } task_d \in Tasks_{Service1}^{mode_i}\right\}$$

3. $$(taskOutPorts_{task_e}{}', taskPvtPorts_{task_e}{}') = f^{task_e}(taskInPorts_{task_e}, taskPvtPorts_{task_e})$$

$$\forall\ task_e \in \Big\{task_e \mid (task_e \in Tasks^{mode_i}_{Service1}) \wedge (t' = t^{switch}_{Service1} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_e}}\}) \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_e}\}\Big\}$$

where

f^{task_e} = The function used in the definition for $task_e$ of CPS service $Service1$

4. $$taskInPorts_{task_f}{}' = LoadTaskInputs^{mode_i}_{task_f}(sensePorts^{mode_i}_{Service1}{}', inMsgPorts^{mode_i}_{Service1}{}', taskOutPorts^{mode_i}_{Service1}{}')$$

$$\forall\ task_f \in \{task_f \mid (task_f \in Tasks^{mode_i}_{Service1}) \wedge (t' = t^{switch}_{Service1} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_f}}\} \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_f}\})\}$$

5. $$actPorts^{mode_i}_{Service1}{}' = TaskOutsToActs^{mode_i}_{Service1}(taskOutPorts^{mode_i}_{Service1}{}')$$

6. $$outMsgPorts^{mode_i}_{Service1}{}' = TaskOutsToOutMsgs^{mode_i}_{Service1}(taskOutPorts^{mode_i}_{Service1}{}')$$

- $\mathbf{\Theta_{P_{1Service}}}$ — An *initial condition.*

Any initial state s of transition system $P_{1Service}$ must satisfy the following initial conditions:

$$t = 0, \qquad t^{switch}_{Service1} = 0$$

$$mode_{Service1} = mode_1$$

$$t^{next}_{Service1} = \min\{t_j \mid (t_j > 0) \wedge (t_j = a\{\tfrac{Period_{mode_1}}{TaskFreq_{task_d}}\} \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_d}\} \text{ and for some } task_d \in Tasks^{mode_1}_{Service1})\ \}$$

APPENDIX C

MANNA-PNUELI TRANSITION SYSTEM REPRESENTATION: CPS COMPUTING NODE WITH K CPS SERVICES

A CPS computing node with k successfully deployed Giotto-based CPS services $(Service1, Service2, \ldots, ServiceK)$ can be represented as the following Manna-Pnueli Transition System, $P_{kServices} < \Pi_{P_{kServices}}, \Sigma_{P_{kServices}}, T_{P_{kServices}}, \Theta_{P_{kServices}} >$:

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

$$\Pi_{P_{kServices}} = \{t, t^{switch}_{Service1}, mode_{Service1}, t^{next}_{Service1}$$

$$sensePort^{1}_{Service1}, sensePort^{2}_{Service1}, \ldots, sensePort^{p}_{Service1},$$

$$inMsgPort^{1}_{Service1}, inMsgPort^{2}_{Service1}, \ldots, inMsgPort^{r}_{Service1},$$

$$actPort^{1}_{Service1}, actPort^{2}_{Service1}, \ldots, actPort^{q}_{Service1},$$

$$outMsgPort^{1}_{Service1}, outMsgPort^{2}_{Service1}, \ldots, outMsgPort^{l}_{Service1},$$

$$taskInPort^{1}_{Service1}, taskInPort^{2}_{Service1}, \ldots, taskInPort^{a}_{Service1},$$

$$taskoutPort^{1}_{Service1}, taskOutPort^{2}_{Service1}, \ldots, taskOutPort^{b}_{Service1},$$

$$taskPvtPort^{1}_{Service1}, taskPvtPort^{2}_{Service1}, \ldots, taskPvtPort^{c}_{Service1},$$

$$t^{switch}_{Service2}, mode_{Service2}, t^{next}_{Service1}$$

$$sensePort^{1}_{Service2}, sensePort^{2}_{Service2}, \ldots, sensePort^{p}_{Service2},$$

$$inMsgPort^{1}_{Service2}, inMsgPort^{2}_{Service2}, \ldots, inMsgPort^{r}_{Service2},$$

$$actPort^{1}_{Service2}, actPort^{2}_{Service2}, \ldots, actPort^{q}_{Service2},$$

$$outMsgPort^{1}_{Service2}, outMsgPort^{2}_{Service2}, \ldots, outMsgPort^{l}_{Service2},$$

$taskInPort^1_{Service2}, taskInPort^2_{Service2}, \ldots, taskInPort^a_{Service2},$

$taskoutPort^1_{Service2}, taskOutPort^2_{Service2}, \ldots, taskOutPort^b_{Service2},$

$taskPvtPort^1_{Service2}, taskPvtPort^2_{Service2}, \ldots, taskPvtPort^c_{Service2}\}$

$\ldots$

$\ldots$

$t^{switch}_{ServiceK}, mode_{ServiceK}, t^{next}_{ServiceK}$

$sensePort^1_{ServiceK}, sensePort^2_{ServiceK}, \ldots, sensePort^p_{ServiceK},$

$inMsgPort^1_{ServiceK}, inMsgPort^2_{ServiceK}, \ldots, inMsgPort^r_{ServiceK},$

$actPort^1_{ServiceK}, actPort^2_{ServiceK}, \ldots, actPort^q_{ServiceK},$

$outMsgPort^1_{ServiceK}, outMsgPort^2_{ServiceK}, \ldots, outMsgPort^l_{ServiceK},$

$taskInPort^1_{ServiceK}, taskInPort^2_{ServiceK}, \ldots, taskInPort^a_{ServiceK},$

$taskoutPort^1_{ServiceK}, taskOutPort^2_{ServiceK}, \ldots, taskOutPort^b_{ServiceK},$

$taskPvtPort^1_{ServiceK}, taskPvtPort^2_{ServiceK}, \ldots, taskPvtPort^c_{ServiceK}\}$

where

t = time,

$t^{switch}_{Service1}$ = latest mode switch time of CPS service, $Service1$,

$mode_{Service1}$ = current mode of CPS service, $Service1$,

$t^{next}_{Service1}$ = next relevant time instant (task update, actuator update, output message update) during the operation of CPS service $Service1$ in its current mode,

$t^{prev}_{Service1}$ = previous relevant time instant (task update, actuator update, output message update) during the operation of CPS service $Service1$ in its current mode,

$sensePort^{i}_{Service1}$ = sensor port of CPS service $Service1$,

$inMsgPort^{i}_{Service1}$ = input message port of CPS service $Service1$,

$actPort^{i}_{Service1}$ = actuator port of CPS service $Service1$,

$outMsgPort^{i}_{Service1}$ = output message port of CPS service $Service1$,

$taksInPort^{i}_{Service1}$ = input port of a task in CPS service $Service1$,

$taskOutPort^{i}_{Service1}$ = output port of a task in CPS service $Service1$,

$taskPvtPort^{i}_{Service1}$ = private port of a task in CPS service $Service1$,

$t^{switch}_{Service2}$ = latest mode switch time of CPS service, $Service2$,

$mode_{Service2}$ = current mode of CPS service, $Service2$,

$t^{next}_{Service2}$ = next relevant time instant (task update, actuator update,
output message update) during the operation of CPS
service $Service2$ in its current mode,

$t^{prev}_{Service2}$ = previous relevant time instant (task update, actuator update,
output message update) during the operation of CPS
service $Service2$ in its current mode,

$sensePort^{i}_{Service2}$ = sensor port of CPS service $Service2$,

$inMsgPort^{i}_{Service2}$ = input message port of CPS service $Service2$,

$actPort^{i}_{Service2}$ = actuator port of CPS service $Service2$,

$outMsgPort^{i}_{Service2}$ = output message port of CPS service $Service2$,

$taksInPort^{i}_{Service2}$ = input port of a task in CPS service $Service2$,

$taskOutPort^{i}_{Service2}$ = output port of a task in CPS service $Service2$,

$taskPvtPort^{i}_{Service2}$ = private port of a task in CPS service $Service2$,

$t^{switch}_{ServiceK}$ = latest mode switch time of CPS service, $ServiceK$,

$mode_{ServiceK}$ = current mode of CPS service, $ServiceK$,

$t^{next}_{ServiceK}$ = next relevant time instant (task update, actuator update, output message update) during the operation of CPS service $ServiceK$ in its current mode,

$t^{prev}_{ServiceK}$ = previous relevant time instant (task update, actuator update, output message update) during the operation of CPS service $ServiceK$ in its current mode,

$sensePort^{i}_{ServiceK}$ = sensor port of CPS service $ServiceK$,

$inMsgPort^{i}_{ServiceK}$ = input message port of CPS service $ServiceK$,

$actPort^{i}_{ServiceK}$ = actuator port of CPS service $ServiceK$,

$outMsgPort^{i}_{ServiceK}$ = output message port of CPS service $ServiceK$,

$taksInPort^{i}_{ServiceK}$ = input port of a task in CPS service $ServiceK$,

$taskOutPort^{i}_{ServiceK}$ = output port of a task in CPS service $ServiceK$,

$taskPvtPort^{i}_{ServiceK}$ = private port of a task in CPS service $ServiceK$.

- $\mathbf{\Sigma_{P_{kServices}}}$ — 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, $t^{switch}_{Service1}$, $t^{next}_{Service1}$, $t^{prev}_{Service1}$, $t^{switch}_{Service2}$, $t^{next}_{Service2}$, $t^{prev}_{Service2}$, $\ldots$, $t^{switch}_{ServiceK}$, $t^{next}_{ServiceK}$, and $t^{prev}_{ServiceK}$ is $\mathbb{R}_{\geq 0}$. The domains of state variables $mode_{Service1}$, $mode_{Service2}$, $\ldots$, and $mode_{ServiceK}$ are $\mathbb{M}_{Service1}$ = {Set of modes of CPS service $Service1$}, $\mathbb{M}_{Service2}$ = {Set of modes of CPS service $Service2$}, $\ldots$, and $\mathbb{M}_{ServiceK}$ = {Set of modes of CPS service $ServiceK$} respectively. Given the following definitions of Π_α and $\mathbb{D}$, all the state variables in Π_α have the domain $\mathbb{D}$:

$$\begin{aligned}\Pi_\alpha = \{&sensePort^i_{Service1}, actPort^i_{Service1}, outMsgPort^i_{Service1},\\ &taskInPort^i_{Service1}, taskOutPort^i_{Service1}, taskPvtPort^i_{Service1},\\ &sensePort^i_{Service2}, actPort^i_{Service2}, outMsgPort^i_{Service2},\\ &taskInPort^i_{Service2}, taskOutPort^i_{Service2}, taskPvtPort^i_{Service2},\\ &\ldots\\ &\ldots\\ &sensePort^i_{ServiceK}, actPort^i_{ServiceK}, outMsgPort^i_{ServiceK},\\ &taskInPort^i_{ServiceK}, taskOutPort^i_{ServiceK}, taskPvtPort^i_{ServiceK}\}\end{aligned}$$

$$\begin{aligned}\mathbb{D} = \{x \mid &(x \in \mathbb{R})\\ &\wedge (x \text{ can be represented by type } \textit{double} \text{ of computer system})\}\end{aligned}$$

The state variables $inMsgPort^i_{Service1}$, $inMsgPort^i_{Service2}$, $\ldots$, and $inMsgPort^i_{ServiceK}$ have the following domain:

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

- $\mathbf{T_{P_{kServices}}}$ — A finite set of transitions.

$$\begin{aligned}T_{P_{kServices}} = \tau_I &\cup T^{ModeSwitches}_{Service1} \cup T^{TimeIncrement}_{Service1} \cup T^{ModeSwitches}_{Service2}\\ &\cup T^{TimeIncrement}_{Service2} \cup \cdots \cup T^{ModeSwitches}_{ServiceK} \cup T^{TimeIncrement}_{ServiceK}\end{aligned}$$

where

$$\tau_I = \text{Idling Transition}$$

$$\begin{aligned}T^{ModeSwitches}_{Service1} = \{\tau^{mode_i mode_j}_{Service1} \mid &\exists \text{ a } \textit{mode switch} \text{ from } mode_i \text{ to } mode_j\\ &\text{in CPS service } Service1\}\end{aligned}$$

$$T^{TimeIncrement}_{Service1} = \{\tau^{mode_1}_{Service1}, \tau^{mode_2}_{Service1}, \ldots, \tau^{mode_M}_{Service1}\}$$

$$\begin{aligned}T^{ModeSwitches}_{Service2} = \{\tau^{mode_i mode_j}_{Service2} \mid &\exists \text{ a } \textit{mode switch} \text{ from } mode_i \text{ to } mode_j\\ &\text{in CPS service } Service2\}\end{aligned}$$

$$T^{TimeIncrement}_{Service2} = \{\tau^{mode_1}_{Service2}, \tau^{mode_2}_{Service2}, \ldots, \tau^{mode_M}_{Service2}\}$$

$$T^{ModeSwitches}_{ServiceK} = \{\tau^{mode_i mode_j}_{ServiceK} \mid \exists \text{ a } \textit{mode switch} \text{ from } mode_i \text{ to } mode_j \text{ in CPS service } ServiceK\}$$

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

As outlined earlier in the summary of Manna-Pnueli Transition System approach, each transition τ can be characterized by an *enabling condition* and a *set of modification statements*. Based on the above mentioned set of transitions T of $P_{kServices}$, all the diligent transitions of $P_{kServices}$ can be completely described through the enabling conditions and modification statements of the following generic transitions: $\tau^{mode_i mode_j}_{Service1}$, $\tau^{mode_i}_{Service1}$, $\tau^{mode_i mode_j}_{Service2}$, $\tau^{mode_i}_{Service2}$, $\tau^{mode_i mode_j}_{ServiceK}$ and $\tau^{mode_i}_{ServiceK}$.

a) $\tau^{mode_i mode_j}_{Service1}$*: Enabling Condition*

$$C_{\tau^{mode_i mode_j}_{Service1}} = (mode_{Service1} == mode_i) \wedge ModeSwitchCondition_{Service1}(t, mode_i, mode_j) \wedge ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j)$$

where

$ModeSwitchCondition_{Service1}(t, mode_i, mode_j)$ = An assertion that returns true if the *guard condition* associated with the *driver* of *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service1$ is true.

$ModeSwitchCheckTime_{Service1}(t, t^{switch}_{Service1}, mode_i, mode_j)$ = An assertion that returns true if $t - t^{switch}_{Service1} = a\{\frac{Period_{mode_i}}{SwitchFreq_{mode_i mode_j}}\}$, for some $a \in \{1, 2, \ldots, SwitchFreq_{mode_i mode_j}\}$.

b) $\tau^{mode_i mode_j}_{Service1}$*: Modification Statements*

1. $mode_{Service1}{}' = mode_j$

2. $t^{switch}_{Service1}{}' = t$

3. $t^{prev}_{Service1}{}' = t$

4. $t^{next}_{Service1}{}' = t + t_{jump}$

   where

   $$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t + t_j = t^{switch}_{Service1}{}' + a\{\tfrac{Period_{mode_j}}{TaskFreq_{task_d}}\}$$

   for some $a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$

   and for some $task_d \in Tasks^{mode_j}_{Service1}$) }

5. $taskOutPorts^{mode_j}_{Service1}{}' =$

   $$ModeSwitchFunction^{mode_i mode_j}_{Service1}(taskOutPorts^{mode_i}_{Service1})$$

   where

   $ModeSwitchFunction^{mode_i mode_j}_{Service1}$ = The function used in the definition of the *driver* associated with the *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service1$

6. $actPorts^{mode_j}_{Service1}{}' = TaskOutsToActs^{mode_j}_{Service1}(taskOutPorts^{mode_j}_{Service1}{}')$

   where

   $TaskOutsToActs^{mode_j}_{Service1}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the actuator ports in $mode_j$ of CPS service $Service1$.

7. $outMsgPorts^{mode_j}_{Service1}{}' =$

   $$TaskOutsToOutMsgs^{mode_j}_{Service1}(taskOutPorts^{mode_j}_{Service1}{}')$$

   where

   $TaskOutsToOutMsgs^{mode_j}_{Service1}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the output message ports in $mode_j$ of CPS service $Service1$.

8. $$taskInPorts_{task_b}{}' = LoadTaskInputs^{mode_j}_{task_b}(sensePorts^{mode_j}_{Service1}{}',$$
$$inMsgPorts^{mode_j}_{Service1}{}', taskOutPorts_{task_b}{}')$$

for every $task_b \in Tasks^{mode_j}_{Service1}$

where

$LoadTaskInputs^{mode_j}_{task_b}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the task input ports of $task_b$ in $mode_j$ of CPS service $Service1$.

c) $\tau^{mode_i}_{Service1}$: Enabling Condition

$$C_{\tau^{mode_i}_{Service1}} = (mode_{Service1} == mode_i)$$
$$\wedge\ (\forall A \in \{1, 2, \ldots, K\}, t^{next}_{Service1} \leq t^{next}_{ServiceA})$$
$$\wedge\ \neg\big(ModeSwitchCondition_{Service1}(t^{prev}_{Service1}, mode_i, mode_c) \wedge$$
$$ModeSwitchCheckTime_{Service1}(t^{prev}_{Service1}, t^{switch}_{Service1}, mode_i, mode_c)\big)$$

$\forall\ mode_c \in \{mode_c \mid \exists$ a *mode switch* from $mode_i$ to $mode_c$ of CPS service $Service1$ $\}$

d) $\tau^{mode_i}_{Service1}$: Modification Statements

1. $t' = t^{next}_{Service1}$

2. $t^{prev}_{Service1}{}' = t'$

3. $t^{next}_{Service1}{}' = t' + t_{jump}$

where

$$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t' + t_j = t^{switch}_{Service1} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_d}}\}$$
$$\text{for some } a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$$
$$\text{and for some } task_d \in Tasks^{mode_i}_{Service1})\ \}$$

4. $(taskOutPorts_{task_e}{}', taskPvtPorts_{task_e}{}') =$

$$f^{task_e}(taskInPorts_{task_e}, taskPvtPorts_{task_e})$$

$$\forall\ task_e \in \{task_e \mid (task_e \in Tasks^{mode_i}_{Service1})$$
$$\wedge\ (t' = t^{switch}_{Service1} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_e}}\}$$
$$\text{for some } a \in \{1, 2, \ldots, TaskFreq_{task_e}\})\}$$

where

f^{task_e} = The function used in the definition for $task_e$ of CPS service $Service1$

5. $taskInPorts_{task_f}{}' = LoadTaskInputs^{mode_i}_{task_f}(sensePorts^{mode_i}_{Service1}{}',$

$$inMsgPorts^{mode_i}_{Service1}{}', taskOutPorts^{mode_i}_{Service1}{}')$$

$$\forall\ task_f \in \{task_f \mid (task_f \in Tasks^{mode_i}_{Service1})$$
$$\wedge\ (t' = t^{switch}_{Service1} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_f}}\}$$
$$\text{for some } a \in \{1, 2, \ldots, TaskFreq_{task_f}\})\}$$

6. $actPorts^{mode_i}_{Service1}{}' = TaskOutsToActs^{mode_i}_{Service1}(taskOutPorts^{mode_i}_{Service1}{}')$

7. $outMsgPorts^{mode_i}_{Service1}{}' =$

$$TaskOutsToOutMsgs^{mode_i}_{Service1}(taskOutPorts^{mode_i}_{Service1}{}')$$

e) $\tau^{mode_i mode_j}_{Service2}$: Enabling Condition

$$C_{\tau^{mode_i mode_j}_{Service2}} = (mode_{Service2} == mode_i)$$

$$\wedge\ ModeSwitchCondition_{Service2}(t, mode_i, mode_j)$$

$$\wedge\ ModeSwitchCheckTime_{Service2}(t, t^{switch}_{Service2}, mode_i, mode_j)$$

where

$ModeSwitchCondition_{Service2}(t, mode_i, mode_j) =$ An assertion that returns true if the *guard condition* associated with the *driver* of *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service2$ is true at time t.

$ModeSwitchCheckTime_{Service2}(t, t^{switch}_{Service2}, mode_i, mode_j) =$ An assertion that returns true if $t - t^{switch}_{Service2} = a\{\frac{Period_{mode_i}}{SwitchFreq_{mode_i mode_j}}\}$, for some $a \in \{1, 2, \ldots, SwitchFreq_{mode_i mode_j}\}$.

f) $\tau^{mode_i mode_j}_{Service2}$: Modification Statements

1. $mode_{Service2}{}' = mode_j$
2. ${t^{switch}_{Service2}}' = t$
3. ${t^{prev}_{Service2}}' = t$
4. ${t^{next}_{Service2}}' = t + t_{jump}$

   where

   $$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t + t_j = {t^{switch}_{Service2}}' + a\{\frac{Period_{mode_j}}{TaskFreq_{task_d}}\}$$
   $$\text{for some } a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$$
   $$\text{and for some } task_d \in Tasks^{mode_j}_{Service2})\ \}$$
5. ${taskOutPorts^{mode_j}_{Service2}}' =$

   $$ModeSwitchFunction^{mode_i mode_j}_{Service2}(taskOutPorts^{mode_i}_{Service2})$$

   where

   $ModeSwitchFunction^{mode_i mode_j}_{Service2} =$ The function used in the definition of the *driver* associated with the *mode switch* from $mode_i$ to $mode_j$ of CPS service $Service2$

6. $actPorts^{mode_j}_{Service2}{}' = TaskOutsToActs^{mode_j}_{Service2}(taskOutPorts^{mode_j}_{Service2}{}')$

where

$TaskOutsToActs^{mode_j}_{Service2}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the actuator ports in $mode_j$ of CPS service $Service2$.

7. $outMsgPorts^{mode_j}_{Service2}{}' =$

$$TaskOutsToOutMsgs^{mode_j}_{Service2}(taskOutPorts^{mode_j}_{Service2}{}')$$

where

$TaskOutsToOutMsgs^{mode_j}_{Service2}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the output message ports in $mode_j$ of CPS service $Service2$.

8. $taskInPorts_{task_b}{}' = LoadTaskInputs^{mode_j}_{task_b}(sensePorts^{mode_j}_{Service2}{}',$

$$inMsgPorts^{mode_j}_{Service2}{}', taskOutPorts_{task_b}{}')$$

for every $task_b \in Tasks^{mode_j}_{Service2}$

where

$LoadTaskInputs^{mode_j}_{task_b}$ = A function that captures input-output relationship (produced by the combined effect) of all the *drivers*, updating the task input ports of $task_b$ in $mode_j$ of CPS service $Service2$.

g) $\tau^{mode_i}_{Service2}$*: Enabling Condition*

$$C_{\tau^{mode_i}_{Service2}} = (mode_{Service2} == mode_i)$$

$$\wedge\ (\forall A \in \{1, 2, \ldots, K\}, t^{next}_{Service2} \leq t^{next}_{ServiceA})$$

$$\wedge\ \neg\big(ModeSwitchCondition_{Service2}(t^{prev}_{Service2}, mode_i, mode_c)\ \wedge$$

$$ModeSwitchCheckTime_{Service2}(t^{prev}_{Service2}, t^{switch}_{Service2}, mode_i, mode_c)\big)$$

$\forall\ mode_c \in \{mode_c \mid \exists$ a *mode switch* from $mode_i$ to $mode_c$ of CPS

service $Service2$ $\}$

h) $\tau^{mode_i}_{Service2}$*: Modification Statements*

1. $t' = t^{next}_{Service2}$

2. ${t^{prev}_{Service2}}' = t'$

3. ${t^{next}_{Service2}}' = t' + t_{jump}$

   where

   $t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t' + t_j = t^{switch}_{Service2} + a\{\frac{Period_{mode_i}}{TaskFreq_{task_d}}\}$

   for some $a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$

   and for some $task_d \in Tasks^{mode_i}_{Service2})$ $\}$

4. $(taskOutPorts_{task_e}{}', taskPvtPorts_{task_e}{}') = f^{task_e}(taskInPorts_{task_e},$

   $taskPvtPorts_{task_e})$

   $\forall\ task_e \in \{task_e \mid (task_e \in Tasks^{mode_i}_{Service2})$

   $\wedge\ (t' = t^{switch}_{Service2} + a\{\frac{Period_{mode_i}}{TaskFreq_{task_e}}\}$

   for some $a \in \{1, 2, \ldots, TaskFreq_{task_e}\})\}$

   where

   f^{task_e} = The function used in the definition for $task_e$ of

   CPS service $Service2$

5. $taskInPorts_{task_f}{}' = LoadTaskInputs^{mode_i}_{task_f}(sensePorts^{mode_i}_{Service2}{}',$
$inMsgPorts^{mode_i}_{Service2}{}', taskOutPorts^{mode_i}_{Service2}{}')$

$\forall\ task_f \in \{task_f \mid (task_f \in Tasks^{mode_i}_{Service2})$
$\wedge\ (t' = t^{switch}_{Service2} + a\{\frac{Period_{mode_i}}{TaskFreq_{task_f}}\}$
for some $a \in \{1, 2, \ldots, TaskFreq_{task_f}\})\}$

6. $actPorts^{mode_i}_{Service2}{}' = TaskOutsToActs^{mode_i}_{Service2}(taskOutPorts^{mode_i}_{Service2}{}')$

7. $outMsgPorts^{mode_i}_{Service2}{}' =$
$TaskOutsToOutMsgs^{mode_i}_{Service2}(taskOutPorts^{mode_i}_{Service2}{}')$

i) $\tau^{mode_i mode_j}_{ServiceK}$*: Enabling Condition*

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

where

$ModeSwitchCondition_{ServiceK}(t, mode_i, mode_j) =$ An assertion that returns true if the *guard condition* associated with the *driver* of *mode switch* from $mode_i$ to $mode_j$ of CPS service $ServiceK$ is true at time t.

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

j) $\tau^{mode_i mode_j}_{ServiceK}$*: Modification Statements*

1. $mode_{ServiceK}{}' = mode_j$

2. $t^{switch}_{ServiceK}{}' = t$

3. $t^{prev}_{ServiceK}{}' = t$

4. $t^{next}_{ServiceK}{}' = t + t_{jump}$

where

$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t + t_j = t^{switch}_{ServiceK}{}' + a\{\frac{Period_{mode_j}}{TaskFreq_{task_d}}\}$

for some $a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$

and for some $task_d \in Tasks^{mode_j}_{ServiceK}$) $\}$

5. $taskOutPorts^{mode_j}_{ServiceK}{}' =$

$ModeSwitchFunction^{mode_i mode_j}_{ServiceK}(taskOutPorts^{mode_i}_{ServiceK})$

where

$ModeSwitchFunction^{mode_i mode_j}_{ServiceK}$ = The function used in the definition of the *driver* associated with the *mode switch* from $mode_i$ to $mode_j$ of CPS service $ServiceK$

6. $actPorts^{mode_j}_{ServiceK}{}' = TaskOutsToActs^{mode_j}_{ServiceK}(taskOutPorts^{mode_j}_{ServiceK}{}')$

where

$TaskOutsToActs^{mode_j}_{ServiceK}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the actuator ports in $mode_j$ of CPS service $ServiceK$.

7. $outMsgPorts^{mode_j}_{ServiceK}{}' =$

$TaskOutsToOutMsgs^{mode_j}_{ServiceK}(taskOutPorts^{mode_j}_{ServiceK}{}')$

where

$TaskOutsToOutMsgs^{mode_j}_{ServiceK}$ = A function that captures the input-output relationship (produced by the combined effect) of all the *drivers*, updating the output message ports in $mode_j$ of CPS service $ServiceK$.

8. $taskInPorts_{task_b}{}' = LoadTaskInputs_{task_b}^{mode_j}(sensePorts_{ServiceK}^{mode_j}{}',$
$inMsgPorts_{ServiceK}^{mode_j}{}', taskOutPorts_{task_b}{}')$

for every $task_b \in Tasks_{ServiceK}^{mode_j}$

where

$LoadTaskInputs_{task_b}^{mode_j}$ = A function that captures input-output relationship (produced by the combined effect) of all the *drivers*, updating the task input ports of $task_b$ in $mode_j$ of CPS service $ServiceK$.

k) $\tau_{ServiceK}^{mode_i}$*: Enabling Condition*

$$\begin{aligned}C_{\tau_{ServiceK}^{mode_i}} = {} & (mode_{ServiceK} == mode_i) \\ & \wedge (\forall A \in \{1, 2, \ldots, K\}, t_{ServiceK}^{next} \leq t_{ServiceA}^{next}) \\ & \wedge \neg\big(ModeSwitchCondition_{ServiceK}(t_{ServiceK}^{prev}, mode_i, mode_c) \wedge \\ & \quad ModeSwitchCheckTime_{ServiceK}(t_{ServiceK}^{prev}, t_{ServiceK}^{switch}, mode_i, mode_c)\big)\end{aligned}$$

$\forall\ mode_c \in \{mode_c \mid \exists$ a *mode switch* from $mode_i$ to $mode_c$ of CPS service $ServiceK\ \}$

l) $\tau_{ServiceK}^{mode_i}$*: Modification Statements*

1. $t' = t_{ServiceK}^{next}$
2. $t_{ServiceK}^{prev}{}' = t'$
3. $t_{ServiceK}^{next}{}' = t' + t_{jump}$

   where

$$t_{jump} = \min\{t_j \mid (t_j > 0) \wedge (t' + t_j = t^{switch}_{ServiceK} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_d}}\}$$

$$\text{for some } a \in \{1, 2, \ldots, TaskFreq_{task_d}\}$$

$$\text{and for some } task_d \in Tasks^{mode_i}_{ServiceK})\ \}$$

4. $(taskOutPorts_{task_e}{}', taskPvtPorts_{task_e}{}') = f^{task_e}(taskInPorts_{task_e}, taskPvtPorts_{task_e})$

$$\forall\ task_e \in \{task_e \mid (task_e \in Tasks^{mode_i}_{ServiceK}) \wedge (t' = t^{switch}_{ServiceK} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_e}}\} \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_e}\})\}$$

where

f^{task_e} = The function used in the definition for $task_e$ of CPS service $ServiceK$

5. $taskInPorts_{task_f}{}' = LoadTaskInputs^{mode_i}_{task_f}(sensePorts^{mode_i}_{ServiceK}{}', inMsgPorts^{mode_i}_{ServiceK}{}', taskOutPorts^{mode_i}_{ServiceK}{}')$

$$\forall\ task_f \in \{task_f \mid (task_f \in Tasks^{mode_i}_{ServiceK}) \wedge (t' = t^{switch}_{ServiceK} + a\{\tfrac{Period_{mode_i}}{TaskFreq_{task_f}}\} \text{ for some } a \in \{1, 2, \ldots, TaskFreq_{task_f}\})\}$$

6. $actPorts^{mode_i}_{ServiceK}{}' = TaskOutsToActs^{mode_i}_{ServiceK}(taskOutPorts^{mode_i}_{ServiceK}{}')$

7. $outMsgPorts^{mode_i}_{ServiceK}{}' = TaskOutsToOutMsgs^{mode_i}_{ServiceK}(taskOutPorts^{mode_i}_{ServiceK}{}')$

- $\mathbf{\Theta_{P_{kServices}}}$ — An *initial condition.*

Any initial state s of transition system $P_{kServices}$ must satisfy the following initial conditions:

$$t = 0$$

$$t^{switch}_{Service1} = 0$$

$t^{switch}_{Service2} = 0$

$\dots$

$\dots$

$t^{switch}_{ServiceK} = 0$

$mode_{Service1} = mode_1$

$mode_{Service2} = mode_1$

$\dots$

$\dots$

$mode_{ServiceK} = mode_1$

$t^{next}_{Service1} = \min\{t_j \mid (t_j > 0) \wedge (t_j = a\{\frac{Period_{mode_1}}{TaskFreq_{task_d}}\}$

for some $a \in \{1, 2, \dots, TaskFreq_{task_d}\}$

and for some $task_d \in Tasks^{mode_1}_{Service1})$ $\}$

$t^{next}_{Service2} = \min\{t_j \mid (t_j > 0) \wedge (t_j = a\{\frac{Period_{mode_1}}{TaskFreq_{task_d}}\}$

for some $a \in \{1, 2, \dots, TaskFreq_{task_d}\}$

and for some $task_d \in Tasks^{mode_1}_{Service2})$ $\}$

$\dots$

$\dots$

$t^{next}_{ServiceK} = \min\{t_j \mid (t_j > 0) \wedge (t_j = a\{\frac{Period_{mode_1}}{TaskFreq_{task_d}}\}$

for some $a \in \{1, 2, \dots, TaskFreq_{task_d}\}$

and for some $task_d \in Tasks^{mode_1}_{ServiceK})$ $\}$

REFERENCES

[1] Albadi, M. H. and El-Saadany, E., "A summary of demand response in electricity markets," *Electric power systems research*, vol. 78, no. 11, pp. 1989–1996, 2008.

[2] Almeida, J. B., Frade, M. J., Pinto, J. S., and de Sousa, S. M., *Rigorous software development: an introduction to program verification.* Springer Science & Business Media, 2011.

[3] Alur, R., Arzen, K.-E., Baillieul, J., Henzinger, T., Hristu-Varsakelis, D., and Levine, W. S., *Handbook of networked and embedded control systems.* Springer Science & Business Media, 2007.

[4] Antsaklis, P. J., "Hybrid control systems: An introductory discussion to the special issue," *Automatic Control, IEEE Transactions on*, vol. 43, no. 4, pp. 457–460, 1998.

[5] Ben-Ari, M., *Mathematical logic for computer science.* Springer Science & Business Media, 2012.

[6] Berry, G. and Gonthier, G., "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[7] Brogan, W. L., *Modern control theory.* Prentice-Hall, New Jersey, 1991.

[8] Brown, A., Carney, D., Feiler, P., Oberndorf, P., and Zelkowitz, M., "A project support environment reference model," in *Proceedings of the conference on TRI-Ada'93*, pp. 82–89, ACM, 1993.

[9] Brown, A. W., Earl, A. N., and McDermid, J., *Software Engineering Environments: Automated Support for Software Engineering.* McGraw-Hill, New York, 1992.

[10] Buttazzo, G., *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24. Springer Science & Business Media, 2011.

[11] Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P., "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," in *ACM Sigplan Notices*, vol. 38, pp. 153–162, ACM, 2003.

[12] Collinson, R. P., *Introduction to avionics systems.* Springer Science & Business Media, 2013.

[13] Costley, M. and Grijalva, S., "Efficient distributed opf for decentralized power system operations and electricity markets," in *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*, pp. 1–6, IEEE, 2012.

[14] Dugard, L. and Verriet, E., "Stability and control of time-delay systems," *Lecture notes in control and information sciences*, 1998.

[15] Egerstedt, M., Wardi, Y., and Axelsson, H., "Transition-time optimization for switched-mode dynamical systems," *IEEE Transactions on Automatic Control*, vol. 51, pp. 110–115, Jan 2006.

[16] Erl, T., *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, New Jersey, 2005.

[17] Feiler, P. H. and Gluch, D. P., *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language.* Addison-Wesley, 2012.

[18] Feizollahi, M. J., Costley, M., Ahmed, S., and Grijalva, S., "Large-scale decentralized unit commitment," *International Journal of Electrical Power & Energy Systems*, vol. 73, pp. 97–106, 2015.

[19] Frankel, D. S., *Model driven architecture: applying MDA to enterprise computing.* Wiley publishing, 2003.

[20] Friedenthal, S., Moore, A., and Steiner, R., *A practical guide to SysML: the systems modeling language.* Morgan Kaufmann, 2014.

[21] Georgakopoulos, D. and Papazoglou, M. P., *Service-oriented computing.* The MIT Press, 2008.

[22] Grijalva, S. and Tariq, M. U., "Prosumer-based smart grid architecture enables a flat, sustainable electricity industry," in *Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES*, pp. 1–6, Jan 2011.

[23] Gronback, R. C., *Eclipse Modeling Project: A Domain-Specific Language Toolkit.* Addison-Wesley Professional, Boston, 2009.

[24] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[25] Henzinger, T., Horowitz, B., and Kirsch, C., "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, pp. 84–99, Jan 2003.

[26] Henzinger, T. A. and Kirsch, C. M., "The embedded machine: Predictable, portable real-time code," *ACM Trans. Program. Lang. Syst.*, vol. 29, Oct. 2007.

[27] Henzinger, T. A. and Sifakis, J., "The embedded systems design challenge," in *FM 2006: Formal Methods*, pp. 1–15, Springer, 2006.

[28] IEEE, "Ieee reliability test system," *Power Apparatus and Systems, IEEE Transactions on*, vol. PAS-98, pp. 2047–2054, Nov 1979.

[29] inc, M., "Simscape r2015b." `http://www.mathworks.com/products/simscape/`, 2016.

[30] Issariyakul, T. and Hossain, E., *Introduction to network simulator NS2*. Springer Science & Business Media, 2011.

[31] Jammes, F., Mensch, A., and Smit, H., "Service-oriented device communications using the devices profile for web services," in *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pp. 1–8, ACM, 2005.

[32] Jobredeaux, R., Herencia-Zapana, H., Neogi, N., and Feron, E., "Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems," in *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pp. 1816–1821, Dec 2012.

[33] Jones, M., Kotsalis, G., and Shamma, J. S., "Cyber-attack forecast modeling and complexity reduction using a game-theoretic framework," in *Control of Cyber-Physical Systems*, pp. 65–84, Springer, 2013.

[34] Jones, M., "Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities," in *Symposium on Operating Systems Principles, 1997. 10*, pp. 198–211, 1997.

[35] Kaiser, R., "Alternatives for scheduling virtual machines in real-time embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, (New York, NY, USA), pp. 5–10, ACM, 2008.

[36] Kang, W., Kapitanova, K., and Son, S. H., "Rdds: A real-time data distribution service for cyber-physical systems," *Industrial Informatics, IEEE Transactions on*, vol. 8, no. 2, pp. 393–405, 2012.

[37] Karsai, G. and Sztipanovits, J., "Model-integrated development of cyber-physical systems," in *Software Technologies for Embedded and Ubiquitous Systems*, pp. 46–54, Springer, 2008.

[38] Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T., "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, 2003.

[39] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, New York, 2011.

[40] Laplante, P. A. and Ovaska, S. J., *Real-Time Systems Design and Analysis: Tools for the Practitioner*. John Wiley and Sons, New Jersey, 2012.

[41] Lee, E. A., "Computing needs time," *Commun. ACM*, vol. 52, pp. 70–79, May 2009.

[42] LeGuernic, P., Gautier, T., Le Borgne, M., and Le Maire, C., "Programming real-time applications with signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[43] Liberzon, D. and Morse, A. S., "Basic problems in stability and design of switched systems," *Control Systems, IEEE*, vol. 19, no. 5, pp. 59–70, 1999.

[44] Liu, I., Reineke, J., Broman, D., Zimmer, M., and Lee, E. A., "A pret microarchitecture implementation with repeatable timing and competitive performance," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 87–93, Sept 2012.

[45] Liu, J. W., *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[46] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[47] Manna, Z. and Pnueli, A., *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[48] Manolios, P. and Papavasileiou, V., "Virtual integration of cyber-physical systems by verification," *AVICPS 2010*, p. 65, 2010.

[49] Martin, J., *Design of real-time computer systems*. Prentice-Hall, Inc., 1967.

[50] mathworks inc, "Embedded coder r2015b." `http://www.mathworks.com/products/embedded-coder/`, 2016.

[51] mathworks inc, "Simulink r2015b." `http://www.mathworks.com/products/simulink/`, 2016.

[52] mathworks inc, "Stateflow r2015b." `http://www.mathworks.com/products/stateflow/`, 2016.

[53] Mesbahi, M. and Egerstedt, M., *Graph theoretic methods in multiagent networks*. Princeton University Press, 2010.

[54] Moustafa, H. and Zhang, Y., *Vehicular networks: techniques, standards, and applications*. Auerbach Publications, 2009.

[55] Nazari, M. H., Costello, Z., Feizollahi, M. J., Grijalva, S., and Egerstedt, M., "Distributed frequency control of prosumer-based electric energy systems," *Power Systems, IEEE Transactions on*, vol. 29, no. 6, pp. 2934–2942, 2014.

[56] Necula, G. C., *Proof-carrying code. design and implementation.* Springer, 2002.

[57] ns 3 Collaboration, "The ns-3 network simulator." `http://www.nsnam.org/`, 2016.

[58] OASIS, "Reference model for service oriented architecture 1.0." `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, 2006.

[59] Ogata, K., *Modern control engineering.* Prentice-Hall, New Jersey, 1997.

[60] powerworld corp, "Powerworld simulator v19." `http://www.powerworld.com/products/simulator/overview`, 2016.

[61] Pretschner, A., Broy, M., Kruger, I. H., and Stauner, T., "Software engineering for automotive systems: A roadmap," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 55–71, IEEE Computer Society, 2007.

[62] Ramachandran, T., Costello, Z., Kingston, P., Grijalva, S., and Egerstedt, M., "Distributed power allocation in prosumer networks," in *Estimation and Control of Networked Systems*, vol. 3, pp. 156–161, 2012.

[63] Regli, W. C., Mayk, I., Dugan, C. J., Kopena, J. B., Lass, R. N., Modi, P. J., Mongan, W. M., Salvage, J. K., and Sultanik, E. A., "Development and specification of a reference model for agent-based systems," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 39, no. 5, pp. 572–596, 2009.

[64] Selic, B. and Gérard, S., *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems.* Elsevier, 2013.

[65] Stahl, T., Völter, M., Bettin, J., Haase, A., and Helsen, S., *Model-driven software development: technology, engineering, management.* John Wiley & Sons, 2006.

[66] Stankovic, J. A. and Rajkumar, R., "Real-time operating systems," *Real-Time Systems*, vol. 28, no. 2-3, pp. 237–253, 2004.

[67] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E., *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, Boston, 2008.

[68] Tariq, M. U., Grijalva, S., and Wolf, M., "Towards a distributed, service-oriented control infrastructure for smart grid," in *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pp. 35–44, April 2011.

[69] Tariq, M. U., Al Faruque, M. A., Grijalva, S., and Wolf, M., "Towards a generic, service-oriented framework for distributed real-time systems.," in *Real-time and Distributed Computing in Emerging Applications (REACTION), 2012 First International Workshop on*, 2012.

[70] Tariq, M. U., Florence, J., and Wolf, M., "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, 2014.

[71] Tariq, M. U., Grijalva, S., and Wolf, M., "A service-oriented, cyber-physical reference model for smart grid," in *Cyber Physical Systems Approach to Smart Electric Power Grid*, pp. 25–42, Springer, 2015.

[72] Tariq, M. U., Nasir, H. A., Muhammad, A., and Wolf, M., "Model-driven performance analysis of large scale irrigation networks," in *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pp. 151–160, IEEE, 2012.

[73] Tariq, M. U., Swenson, B. P., Narasimhan, A. P., Grijalva, S., Riley, G. F., and Wolf, M., "Cyber-physical co-simulation of smart grid applications using ns-3," in *Proceedings of the 2014 Workshop on ns-3*, p. 8, ACM, 2014.

[74] Tarraf, D. C., "Control of cyber-physical systems," *Proc. of Lecture Notes in Control and Information Sciences*, vol. 449, 2013.

[75] Tomsovic, K., Bakken, D. E., Venkatasubramanian, V., and Bose, A., "Designing the next generation of real-time control, communication, and computations for large power systems," *Proceedings of the IEEE*, vol. 93, no. 5, pp. 965–979, 2005.

[76] Tsai, W. T., "Service-oriented system engineering: a new paradigm," in *Service-Oriented System Engineering (SOSE), 2005. IEEE International Workshop*, pp. 3–6, 2005.

[77] Varga, A. and Hornig, R., "An overview of the omnet++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, p. 60, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[78] Wan, Y., "A primer on wind power for utility applications," tech. rep., National Renewable Energy Laboratory, Golden, CO./TP-500-36230, 2005.

[79] Wolf, W., "Cyber-physical systems," *Computer*, vol. 42, pp. 88–89, Mar. 2009.

[80] world economic forum, "Accelerating successful smart grid pilots." `http://www3.weforum.org/docs/WEF_EN_SmartGrids_Pilots_Report_2010.pdf`, 2010.

[81] Zhang, F., Szwaykowska, K., Wolf, W., and Mooney, V., "Task scheduling for control oriented requirements for cyber-physical systems," in *Real-Time Systems Symposium, 2008*, pp. 47–56, IEEE, 2008.

[82] Zhang, W., Branicky, M. S., and Phillips, S. M., "Stability of networked control systems," *Control Systems, IEEE*, vol. 21, no. 1, pp. 84–99, 2001.

[83] Zimmermann, H., "Osi reference model–the iso model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

VITA

Muhammad Umer Tariq (BSEE 2007, MSECE 2010) is a PhD Candidate in the Cyber-Physical Systems (CPS) Laboratory at Georgia Tech. He received his B.Sc. in Electrical Engineering from the University of Engineering and Technology in Lahore, Pakistan in 2007. After winning the Fulbright scholarship to pursue his graduate studies in the United States, he earned his M.S. in Electrical and Computer Engineering from Georgia Tech in 2010. In his doctoral studies, he has focused on the development of a service-oriented computing infrastructure for efficient as well as reliable development and operation of wide-area cyber-physical systems such as smart power grid, vehicular networks, and automated irrigation networks. During his doctoral studies, he won the GE Smart Grid Research Challenge for his research project titled "Cloud Computing based Demand Response Solution for Residential Consumers". During his doctoral studies, he was the lead software developer for a simulation-based smart grid test-bed, employed in a research project funded by the Green Electricity Network Integration (GENI) program of Advanced Research Projects Agency for Energy (ARPA-E). He also completed his Graduate Certificate in Engineering Entrepreneurship from Scheller College of Business at Georgia Tech in 2015. He is also the co-founder of a renewable energy start-up company Prosumer-Grid, Inc.