MODELS AND ALGORITHMS FOR CYBER-PHYSICAL SYSTEMS

by

SUMEET GUJRATI

B.Sc., Devi Ahilya University Indore, India, 2002

M.C.A., Indian Institute of Technology Roorkee, India, 2005

M.S., Kansas State University, USA, 2008

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2013

Abstract

In this dissertation, we propose a cyber-physical system model, and based on this model, present algorithms for a set of distributed computing problems. Our model specifies a cyber-physical system as a combination of cyber-infrastructure, physical-infrastructure, and user behavior specification. The cyber-infrastructure is superimposed on the physical-infrastructure and continuously monitors its (physical-infrastructure's) changing state. Users operate in the physical-infrastructure and interact with the cyber-infrastructure using hand-held devices and sensors; and their behavior is specified in terms of actions they can perform (*e.g.*, move, observe). While in traditional distributed systems, users interact solely via the underlying cyber-infrastructure, users in a cyber-physical system may interact directly with one another, access sensor data directly, and perform actions asynchronously with respect to the underlying cyber-infrastructure. These additional types of interactions have an impact on how distributed algorithms for cyber-physical systems are designed. We augment distributed mutual exclusion and predicate detection algorithms so that they can accommodate user behavior, interactions among them and the physical-infrastructure. The new algorithms have two components — one describing the behavior of the users in the physical-infrastructure and the other describing the algorithms in the cyber-infrastructure. Each combination of users' behavior and an algorithm in the cyber-infrastructure yields a different cyber-physical system algorithm. We have performed extensive simulation study of our algorithms using OMNeT++ simulation engine and UPPAAL model checker. We also propose Cyber-Physical System Modeling Language (CPSML) to specify cyber-physical systems, and a centralized global state recording algorithm.

MODELS AND ALGORITHMS FOR CYBER-PHYSICAL SYSTEMS

by

SUMEET GUJRATI

B.Sc., Devi Ahilya University Indore, India, 2002

M.C.A., Indian Institute of Technology Roorkee, India, 2005

M.S., Kansas State University, USA, 2008

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2013

Approved by:

Major Professor
Gurdip Singh

Copyright

Abstract

In this dissertation, we propose a cyber-physical system model, and based on this model, present algorithms for a set of distributed computing problems. Our model specifies a cyber-physical system as a combination of cyber-infrastructure, physical-infrastructure, and user behavior specification. The cyber-infrastructure is superimposed on the physical-infrastructure and continuously monitors its (physical-infrastructure's) changing state. Users operate in the physical-infrastructure and interact with the cyber-infrastructure using hand-held devices and sensors; and their behavior is specified in terms of actions they can perform (*e.g.*, move, observe). While in traditional distributed systems, users interact solely via the underlying cyber-infrastructure, users in a cyber-physical system may interact directly with one another, access sensor data directly, and perform actions asynchronously with respect to the underlying cyber-infrastructure. These additional types of interactions have an impact on how distributed algorithms for cyber-physical systems are designed. We augment distributed mutual exclusion and predicate detection algorithms so that they can accommodate user behavior, interactions among them and the physical-infrastructure. The new algorithms have two components — one describing the behavior of the users in the physical-infrastructure and the other describing the algorithms in the cyber-infrastructure. Each combination of users' behavior and an algorithm in the cyber-infrastructure yields a different cyber-physical system algorithm. We have performed extensive simulation study of our algorithms using OMNeT++ simulation engine and UPPAAL model checker. We also propose Cyber-Physical System Modeling Language (CPSML) to specify cyber-physical systems, and a centralized global state recording algorithm.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Abbreviations

AT Acquire Time

CPS Cyber-Physical System

CPSML Cyber-Physical System Modeling Language

CyS Cyber-subystem

Int Interactions

KRL k-Reverse Link

NH Number of Hops

NM Number of Messages

PAT Physical Area Tree

PhyS Pysical-subystem

SPRA Shortest Path Resource Allocation

TDS Traditional Distributed System

WSN Wireless Sensor Networks

Nomenclature

A A physical area

AE Set of active entities

A_i A physical area with Id i

C A cyber entity

CE Set of cyber entities

C_i A cyber entity with Id i

E Set of communication links in cyber graph

E_{ij} A communication edge between cyber entity C_i and C_j

G_C Cyber graph

G_P Spatial model of CPS

GS Global state of TDS

GS_C Global state $PhyS$ as maintained in CyS

GS_P Global state of $PhyS$

PA Set of physical areas in flat spatial model

PE Set of physical entities

P_i Process with Id i

R A resource

RE Set of reachability edges

Ri A resource with Id i

R_{ij} A reachability edge between areas A_i and A_j

rs A resource

RS Set of resources

S_d Sensing delay or cyber-subsystem delay

Ui User or Active Entity or Physical Entity with Id i

Acknowledgments

This work would not have been possible without the guidance and assistance of several individuals who in one way or another extended their support in completing this work. First and foremost, I would like express my sincere gratitude to my major professor Dr. Gurdip Singh for supervising this work. He introduced me to the field of distributed and cyber-physical systems and provided me useful suggestions to carry out my research. Throughout the years, he supported me with his patience and knowledge whilst providing me the funding to work in my own way. Without him, this dissertation would not have been completed or written.

I am thankful to Dr. Masaaki Mizuno, Dr. Torben Amtoft, Dr. Sanjoy Das and Dr. Sarah Reznikoff for serving as my committee members. Their suggestions during and after proposal were very helpful.

I am thankful to all the professors who taught me and CIS staff for providing me all the facilities.

I am thankful to my friends who supported and encouraged me during my studies. I would like to express my gratitude to my parents Mr. Santosh Gujrati and Mrs. Sadhna Gujrati for their encouragement and love. At the end, I thank my beloved wife, Sonal, for always being there for me, having faith in me, for her support, encouragement and love.

Dedication

To

my daughter Nitya

Chapter 1

Introduction

The focus of this chapter is to provide an introduction to the field of *cyber-physical systems* (CPS) and discuss how they differ from *traditional distributed systems* (TDS). The chapter also highlights traditional distributed computing problems which we will address in the context of CPSs in this dissertation.

1.1 What is CPS?

CPSs are intelligent systems that are often formed as an integration of a computational or cyber system (such as a set of computational processes running on computing platforms that may be equipped with sensors) with a physical system. Both systems in a CPS may be tightly coupled with cyber system continuously sensing the changing state of the physical system. The field of CPS is relatively new and has paved ways to develop intelligent applications in variety of domains, including but not limited to military, healthcare, autonomous cars, smart buildings, and aviation. Each domain has its own domain specific design requirements and challenges which CPS developers may have to address. For example, a CPS for a smart hospital may involve humans in the system, and its developers may have to address how the actions performed by the humans interact with and impact the compu-

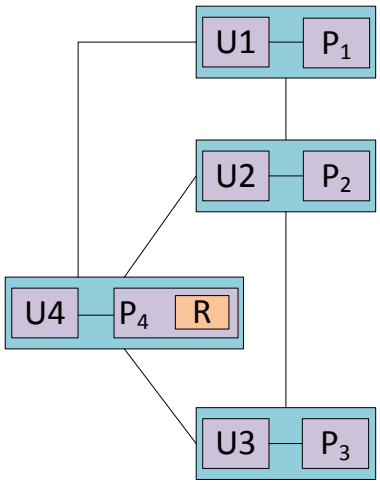
tations performed by the cyber system. These issues, if addressed adequately, can enhance the capabilities of the physical system in a way that the tasks which are accomplished using traditional techniques (perhaps manually) can be performed more efficiently.

In this dissertation, we focus on the CPSs in which actions and behavior of users affect the way the cyber system performs computations. The type of CPSs which we study in this work builds on the existing field of wireless sensor networks (WSN), which is a collection of physically distributed computing nodes equipped with sensors. The mission of an application running on computing nodes in a WSN is to cooperatively monitor physical phenomena (*e.g.*, temperature, pressure) in a specific region and transfer related data via wireless means to a central location, and to control actuators based on the sensed information. A WSN implements traditional distributed algorithms to solve a problem, which at the same time, tries to optimize energy consumption of the computing nodes.

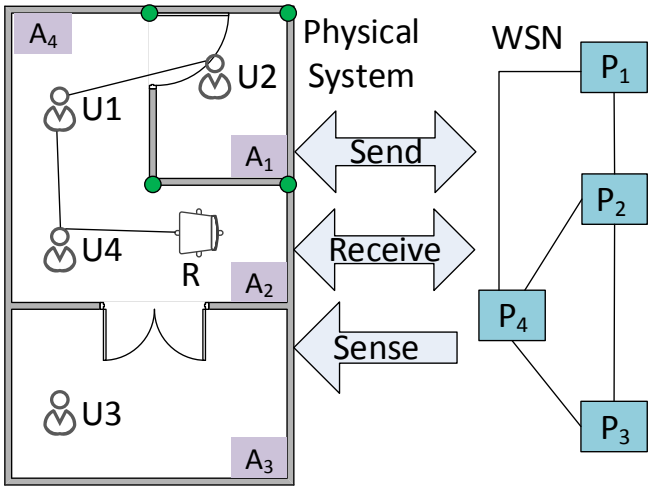
A typical CPS is shown in Figure 1.1(b) in which a cyber system (typically a WSN) is superimposed on a physical system. The CPS application hosted on a WSN implements the computing and networking processes. In order for WSN algorithms to be usable for CPSs, they must also contend with users' behavior and their interactions. How a stand alone WSN, which is modeled as a TDS differs from a CPS is discussed in the next section.

1.2 TDS Model versus CPS Model

In a WSN, which is typically modeled as a TDS, a strict layered approach is taken wherein a set of users (application processes) $U1, \ldots, Un$ is layered on top of a traditional distributed algorithm running as processes $P_1, \ldots, P_n$ (see Figure 1.1(a)). User Ui interacts with process P_i to request a service (such as a resource abstracted as a token), and users rely entirely on the algorithm, *i.e.*, the users in a TDS operate in a controlled environment regulated by the algorithm. In a CPS, a WSN is superimposed over a physical system and the users (typically humans) may themselves possess capabilities such as sensing, observing and



(a) A TDS model: users interact only with dedicated processes and resources are abstracted as tokens



(b) A CPS model: users also interact with each other and act on their own (shown as edges from a user to other users and physical resources)

Figure 1.1: TDS *model versus* CPS *model*

mobility using which they may also attempt to modify the state of the physical system (see Figure 1.1(b)). Thus, the users in a CPS operate in an uncontrolled environment where they also directly interact with each other. The fact that users can act independently of the WSN in a CPS forms the crux of our work as discussed in the next section.

1.3 Problem Definition

There has been considerable work done to define models and extend traditional algorithms in the context of a WSN which is a step in the direction towards CPS. Although one can use traditional algorithms developed for a WSN in a CPS, they appear on one end of the spectrum where all actions are controlled entirely by the WSN. In a CPS, however, we may want to allow the possibility of users in the physical system and processes in the WSN to cooperate/interact in solving a problem. These enhanced capabilities in a CPS provide an opportunity to design more efficient solutions to distributed computing problems. We highlight this by elaborating two problems.

1.3.1 Mutual Exclusion

Consider a problem in a health care facility (see Figure 1.2 [1]) in which users (*e.g.*, hospital staff) may need access to shared resources (*e.g.*, wheelchairs) located in different parts of the facility. In traditional systems, users may rely on established rules wherein free resources are deposited at a central or a set of known locations. This solution is at one extreme end in which users locate resources own their own without the help of a cyber system. Whenever a user needs a resource, (s)he can move to the nearest known location and try to find a free resource. Of course, that location may not have a free resource and the user will

[1] In this dissertation, we frequently refer to Figure 1.2 which depicts the first floor of a hospital's CPS. The physical areas are given names such as $WaitingArea1$ and RR (rest room). The hospital is given a name $Hospital1$. The diamond with two vertical bars and a number represents a cyber entity C with its ID, and a shaded area A surrounding a it indicates its sensing range. A Curve between two sensed areas represents the fact that physical entities can move between those two areas.

have to move to another location. This can be made efficient by instrumenting the facility with a WSN and instrumenting the resources with sensing devices to track their location and usage. Now consider other end of extreme in which WSN implements a traditional algorithm and users always consult the algorithm before performing an action. When a user needs a resource, (s)he first requests the algorithm to locate a free resource and waits for the response. On receiving the location of a free resource, (s)he can move to the location of the resource and acquire it. After being used, the resource is released at some location in the physical system.

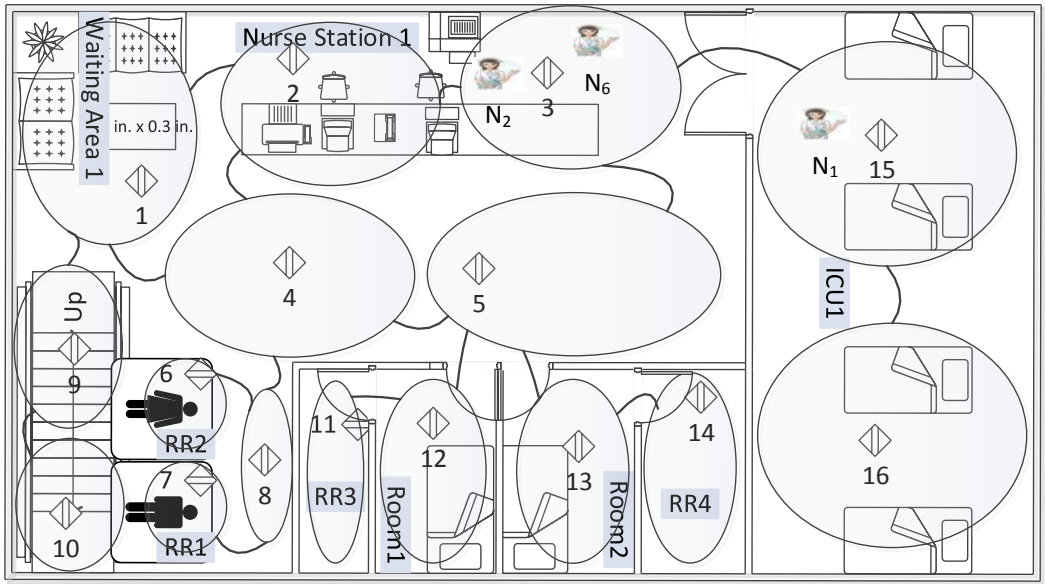


Figure 1.2: *First floor of a hospital's* CPS

In between these two extremes, more efficient solutions may exist that allow the users to perform additional actions independently of WSN. For example, users may have the capability to sense (visually) a free resource on its own, which can be used as follows: to locate a resource, the user sends a request to and awaits response from the WSN. WSN sends a response containing the location of a free resource, say R_1. On receiving this response, the user starts moving towards the location of R_1, and while moving to that location, if the user locates another free resource, say R_2, on its own, then it can acquire it. From the user's point of view, this is more efficient as it is able to get a resource (R_2) earlier

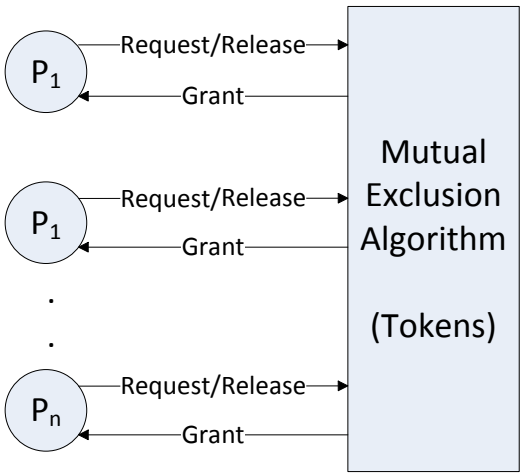
than it would have otherwise (as R_1 may have been further away). However, the algorithm running on the WSN, which is usually implemented as traditional algorithm (centralized or distributed) should be able to adapt to the actions of the users — in particular, it should be able to sense that the user acquired R_2 instead of R_1, and should include R_1 and exclude R_2 from future searches. Clearly, this solution falls in the domain of CPS.

This problem of sharing of physical resources in a CPS is analogous to the problem of mutual exclusion in a TDS in which processes request to obtain exclusive access to resources which are represented as tokens [1–9]. If a process makes a request and a token is available, the mutual exclusion algorithm grants the token to the requesting process. After using the resource, the process releases the token to the mutual exclusion algorithm. In a CPS, the resources may be physical entities such as X-Ray carts and wheelchairs, and instead of processes, physical users (*e.g.*, nurses, doctors) request access to those resources. Figure 1.3 shows a comparison of TDS and CPS view of mutual exclusion.

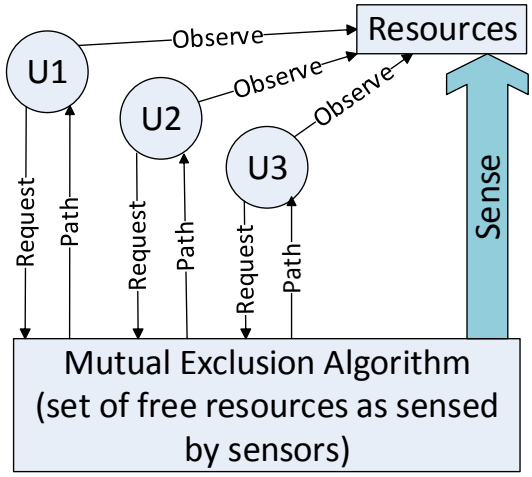
1.3.2 Predicate Detection

There are many instances in which actions of the users in the physical system may have to be constrained to ensure their safe operations. For example, consider the task of transferring a patient from one location to another. A patient may be required to be accompanied by medical staff with specific medical equipment during the transfer. If manual techniques are followed, this constraint is hard to detect and enforce. On the other hand, if a solution based on a traditional distributed algorithm is used (with stand alone WSN), it may force the users to obtain permission from the algorithm before performing any action. However, a CPS solution can be devised in which the algorithms running on the WSN provide the capability of enforcing such constraints by automatically detecting the presence of appropriate medical staff and equipment in the vicinity of the patient at all times during the movement, and issuing alerts when these conditions are violated.

This problem of enforcing constraints and detecting their violation in a CPS is analogous



(a) TDS view of mutual exclusion: users solely rely on the underlying system



(b) CPS view of mutual exclusion: users can independently observe resources and can use them without interacting with the cyber infrastructure

Figure 1.3: *Traditional versus* CPS *view of mutual exclusion*

to the problem of predicate detection in a TDS. The problem of predicate detection in a TDS arises in contexts such as designing, testing and debugging distributed programs. For example, one basic command in a debugging system is *terminate the program if predicate p is true*, which requires detection of predicate p. Similarly, in a CPS, predicate detection algorithms can help in detecting violation of constraints (predicates) in the physical system. CPS can also influence the behavior of the users by providing additional information to them so that they do not perform any action which may cause predicate violation.

Apart from additional interactions between the WSN and the physical system, and enhanced capabilities of users, there are various other issues which need to be addressed when extending traditional distributed algorithms for a CPS. These issues compound the process of integrating the dynamics of the physical system with those of traditional algorithms running on WSN, and provides numerous opportunities of research in the field of CPS. We will discuss some of these issues in Section 2.2. In the following, we briefly discuss the mechanisms that will enable us to address the overall problem of extending traditional distributed algorithms for CPSs.

- The role of the physical system which is monitored by the WSN is integral to the overall behavior of a CPS. Hence, we need to develop a suitable model that captures spatial properties of the physical system.

- We need to model the interactions between the WSN and the physical system, and the additional capabilities of the users, presence of which makes existing algorithms un-suitable for CPSs.

- Once the physical system, interactions and actions can be modeled, we need to develop algorithms executing on WSN to solve the problems in hand (*e.g.*, mutual exclusion and predicate detection).

1.4 Thesis organization

Conventionally, an entire chapter is devoted to literature review — however, we follow a different approach wherein related work is dispersed throughout the dissertation. This dissertation broadly covers four aspects of CPSs: modeling, mutual exclusion algorithms, predicate detection algorithms, and global state recording algorithms. For sake of clarity and ease of understanding and mapping related work in the context of our approach in each of these aspects, we discuss related work in the respective chapters.

The rest of the dissertation is organized as follows. Chapter 2 discusses shortcomings of traditional techniques and presents our approach to address those shortcomings. We also present examples of various domains where our proposed techniques can be applied. The list of domains we present is not exhaustive; however we believe that it does serve the purpose of convincing the reader that our techniques are applicable in variety of domains. This chapter also discusses the work that others have done in the field of CPS.

In Chapter 3, we present a graph based model of CPS which models both the cyber and physical systems. We further propose two models for a physical system, one is a flat spatial model and other is a hierarchical spatial model. We also present *Cyber-Physical System Modeling Language* (CPSML) which provides constructs to model CPSs.

Chapter 4 presents the problem of mutual exclusion in CPSs. We define the problem and propose a centralized and two distributed solutions. We also present exhaustive OMNeT++ based simulation results of our proposed algorithms.

In Chapter 5, we discuss predicates in CPSs, and presents Uppaal based CPS model checker. We also present an algorithm which continuously monitor the CPS, and take proactive actions to minimize predicate violations.

In Chapter 6, we discuss the problem of global state recording in CPSs, and a centralized algorithm to tackle the problem. We conclude the dissertation in Chapter 7.

Chapter 2

Motivation and Approach

We begin this chapter with motivational examples from various domains which can benefit from employing a CPS into them. Following this, we discuss the challenges faced in implementing traditional mutual exclusion, predicate detection, and global state recording algorithms for a CPS, and an overview of our approach addressing these challenges. We conclude this chapter with the related work which has been done in recent years in the field of CPS.

2.1 Motivation

In this section, we motivate our work by discussing problems of mutual exclusion and predicate detection from the domain of parking lot and factory.

2.1.1 Parking Lot

Parking lot is a classic example of problem of sharing physical resources, in which drivers need exclusive access to parking spaces. Without a CPS, drivers look for free parking spaces by driving through the parking lot and at the same time observing it. Drivers may easily overlook an empty parking space during peak hours. Moreover, if a driver finds an empty

space, there are chances that someone else has also observed that particular space and is approaching towards it to use it. If a CPS is employed to address this problem, the drivers can receive assistance from CPS. A driver can request for nearest empty parking space available, and can act on its own to locate one. The driver can be dynamically updated about the usage of various parking spaces. For example, the driver receives the information from the CPS that a parking space is available at first floor. While (s)he drives to the first floor, someone vacated a parking space which is nearer. The CPS can notify the driver of this available parking space.

There are several examples of constraints that may be violated in a parking lot. For example, someone may park a vehicle in reserved parking space not intended for that person, or use the space for a duration longer than allowed. Traditionally, such violations are detected using manual techniques such as the following: an employee may walk around in the parking lot and identify vehicles which are illegally parked, and issue a citation for such vehicles. Not only is such a technique resource intensive, it is inefficient in that many illegally parked vehicles can be overlooked by the staff. Moreover, it is not possible to manually monitor the vehicles at all times. A CPS would not only help to identify such violation instantaneously, but it can also help ensuring more complex constraints, such as *parking lot number 10 must not be occupied by those who do not have Type-S permit between 6AM to 6PM*, and *there must be at least 15 parking lots available at the first floor all the time*.

2.1.2 Factory

Resource sharing is commonly practiced in a factory environment where workers may need exclusive access to tools that are located in various locations in the factory. This problem is solved using manual techniques in a way similar to the problem of locating and using resources in a healthcare facility as discussed in Section 1.3.1. Such a manual technique will introduce similar problems such as a worker may not know which direction to move to

locate a free resource, or two workers may concurrently locate the same free resource and move to that location in which case only one of them will be able to use it.

Furthermore, working conditions in factories may be hazardous. For the safety of workers, their actions may have to be constrained. For example, *there must always be at least N units of resource R_1, at least M units of resource R_2, 3 machine operators, and 2 quality control officers available in an area where product Prod is being produced*, and *concentration of a hazardous gas must not exceed a certain threshold*. Detecting violation of such constraints as soon as they occur is critical in some cases. Delay may prove to be fatal as it happened during Bhopal Disaster [10]. The leakage of poisonous gas affected over 500,000 people and claimed more than 3,000 lives. A CPS could have detected it in timely manner and the tragedy could have been avoided. A CPS employed in a factory would not only help in enforcing constraints and detecting their violations, but would also produce historical data which can later be analyzed to identify usage pattern of certain resources, and working habits of different workers.

2.2 Challenges

In the following, we discuss challenges to be addressed in order to solve the mutual exclusion, predicate detection, and global state recording algorithms for a CPS:

2.2.1 User Behavior

In a TDS, access to a resource is typically controlled solely by a mutual exclusion algorithm (see Figure 1.3(a)). In a CPS, however, the users may not be passive entities – that is, in addition to requesting the WSN to locate resources, the users may themselves actively look for resources (see Figure 1.3(b)). For example, in Figure 1.1(b), in addition to asking the WSN to locate a resource, if $U1$ observes that resource R is available, it may acquire R without waiting for a response from the WSN, something $U1$ would have done if it was in

a purely physical system. This, for instance, may cause scenarios wherein a user, say $U3$ in Fig. 1.1(b), may start using R even though the mutual exclusion algorithm may think that R is free and may concurrently reserve it for another user (as there may not be a way to "lock" a physical resource). Similarly, in a parking lot, a driver may use a parking space without any notification from the WSN, or may use a parking space which WSN might have reserved for someone else. Similar scenarios may also occur in a factory environment.

Predicate detection, another fundamental problem in traditional distributed computing, has various applications such as distributed debugging and monitoring of distributed programs. The goal of predicate detection in such systems is to detect various properties of the distributed system, such as deadlock and termination. The predicates are defined in terms of variables of the processes constituting the system, and the values of these variables are solely controlled by the underlying distributed system. In a CPS, predicates are used to constrain the actions of the users in the physical system. For example, consider a predicate in parking lot which states that *a driver may not park in parking lot number 10 without a valid parking permit between 6AM to 6PM*, and hence if a driver violates this, a warning should be generated. Consider a predicate in a hospital — *there must always be a nurse in the ICU if there is one or more patients in the ICU*. If at any point of time, there is at least one patient and no nurse in the ICU, a warning should be triggered. We notice from these examples that in a CPS, apart from the variables of the processes, the predicates also involve spatial and temporal elements. Moreover, their evaluation may also be influenced by the actions of the users.

2.2.2 Mobility of Resources

In general, a physical resource may be mobile so that it may be acquired at one location and released at another (*e.g.*, a wheelchair may be freed at a different location). This is different from the view taken in a TDS where a resource (*e.g.*, abstracted as a token) is released at the same node where it was acquired, and the traditional distributed algorithms rely on

this property. Hence, they cannot be directly applied for situations where the resource is mobile. Mobility of resources may also lead to violation of predicates. For example, in a hospital, a constraint requiring the presence of at least one X-Ray cart in the operation theater may be violated if the cart is moved to another location. Hence, predicate detection algorithms may react to such actions in the physical system to properly detect violations. This is different from traditional algorithms where the processes react to changes that they themselves make to locally declared variables.

2.2.3 State of a Physical System

In a CPS, the WSN may keep track of the state of the physical system via a set of variables. For example, we may use $Loc.Nurse$ to represent the set of nurses present in a location Loc (*e.g.*, in Figure 1.2, $NurseStation1$ contains two locations A_2 and A_3 monitored separately, and $NurseStation1.Nurse = \{N_2, N_6\}$). Due to sensing delays or lost sensor values, there may be a difference between the actual state of the physical system and the corresponding variables maintained in the WSN. For instance, assume that nurse N_6 moves from A_3 to A_2 at time t. Due to sensing delays, $A_2.Nurse$ may not reflect the presence of N_6 until a later time $t + \delta$. Such delays may result in inconsistencies when applying traditional algorithms directly to a CPS. For example, a traditional global state recording algorithm would form a global state by collecting states of individual areas. In a CPS, sensing delays and unsynchronized clocks may lead to a situation where both A_2 and A_3 report presence of the same nurse. Similarly, we must contend with an opposite scenario where both stations report absence of N_6. This may be caused due to unsynchronized sampling of states or due to the fact that N_6 may be in an intermediate area which is out of the sensing ranges of both A_2 and A_3.

2.2.4 Spatial and Temporal Dependence

A CPS may have spatial consistency constraints that specify conditions on the state of the system based on locations. For example, locations may have a hierarchy associated with them (e.g., $ICU1$ is contained in $Floor1$, and $Floor1$ is contained in $Hospital1$), and one must ensure consistency of their corresponding state variables. For example, in Figure 1.2, since N_1 is *located* in $ICU1$, it counts towards $ICU1.Nurse$; in addition, it must also be included in $Floor1.Nurse$ and $Hospital1.Nurse$.

Furthermore, predicates in a CPS may also have a notion of time associated with them. Examples of such predicates include *C1: if there is one or more patients in ICU, then there must be at least one nurse in ICU*, and *C2: if a patient presses an emergency button, then a nurse should attend the patient within one minute.*

2.3 Overview of Our Approach

Most traditional distributed algorithms are designed with the underlying TDS modeled as a graph. In such a model, nodes represent processes and edges represent communication links between them (see Figure 1.1(a)). In a similar way, to address distributed problems in CPSs, we propose a graph based model which views a CPS as a triple = (*CyS,PhyS,Int*), where *CyS* models the WSN, *PhyS* models the physical system, and *Int* captures the interactions between them. We call *CyS* and *PhyS* the cyber- subsystem and the physical-subsystem respectively. *CyS* is typically a WSN and modeled using a cyber graph G_C, which is similar to a graph used to model a TDS. *PhyS* is modeled using physical graph G_P in which nodes represent physical areas and edges represent reachability edges between them. A reachability edge between two physical areas implies that a user can move directly between those areas. G_C and G_P are modeled such that each node C in G_C has exactly one corresponding matching node A in G_P, implying that each node C monitors exactly one physical area A. A graph based model of CPS shown in Figure 1.1(b) is depicted

in Figure 2.1. In the figure, left half represents G_P and right half represents G_C which is similar to a graph based TDS. There are four nodes (areas), and four reachability edges (shown as curves) in G_P. A curved arrow from a node C in G_C to a node A in G_P represents monitoring of A by C.

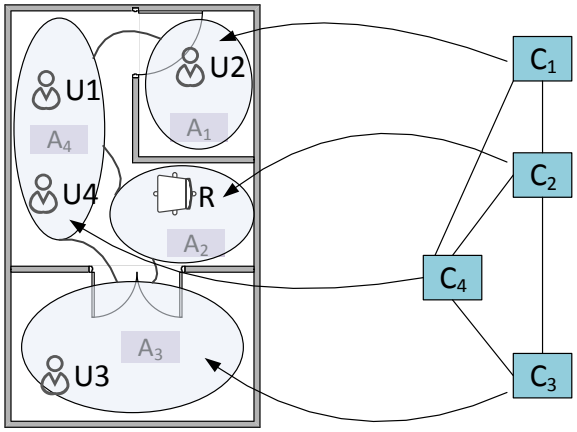


Figure 2.1: *A graph model of a* CPS

For the mutual exclusion problem, we propose a set of three algorithms based on the proposed graph based CPS model. Each algorithm consists of two interacting components, one describing the behavior of the users in *PhyS* and the other describing the mutual exclusion algorithm (cyber algorithm) in *CyS*. Each combination of user behavior and cyber algorithm yields a different CPS algorithm. The behavior of the users range from a simple one (users always follow what *CyS* suggests) to a complex one (users request assistance from *CyS*, but they may also act on their own).

For the global state recording and predicate detection problems, we extend *PhyS* to allow specification of location types/instances and hierarchies similar to [11]. For example, in Figure 1.2, ICU is a logical area (type) and $ICU1$ is an instance of ICU. Each physical area which is monitored by one sensor is a fine grained area. A collection of fine grained areas is called a coarse grained area. For example, $ICU1$ is a coarse grained area consisting of two fine grained areas. Based on this extended CPS model, we propose predicate detection

and model checking techniques for CPS. Predicate detection algorithm is dependent on the nature of the constraints, and is both reactive and proactive in nature. For example, to enforce constraint *C1* specified earlier, the algorithm

- reacts and generates an alert when it detects predicate violation, and
- proactively alerts the nurse from doing actions which may violate *C1*; that is, if only on nurse is present in $ICU1$, then it can alert the nurse that its action of leaving $ICU1$ may violate the constraint. If the nurse's behavior is such that she honors this information, then chances of predicate violation are minimized.

To accommodate specification of temporal inconsistencies between the state of a physical area and its associated variable in *CyS*, we associate a confidence function with each variable (such as with $ICU1.N$), which indicates the extent of accuracy of the current value of that variable.

2.4 Related Work

We find that the problems that arise due to possibilities of additional interactions in CPSs as discussed in Section 2.2, can have a significant impact on the design of distributed algorithms. In [12], interactions between user entities were attributed to "hidden channels" or channels external to the system and possible solutions were proposed which involve the user entities providing additional information such as timestamps. In CPSs, however, user entities may be unable to participate in such timestamping algorithms as they represent actual physical objects. The event model proposed in [13] captures interactions between *PhyS* and *CyS* and defines events based on temporal and spatial attributes, and is a step in providing a theoretical basis to develop CPS distributed algorithms.

There has been some work done that has investigated development of CPS algorithms to address interactions between *PhyS* with *CyS* in several other domains that include vehicular

networks [14–16], water distribution networks [17], smart grid [18, 19], and gardening [20]. We review of this work in the following.

2.4.1 CPS Vehicular Networks

Cyber-physical vehicular networks are designed to assist drivers in safe operation of vehicles as well as to provide current information about surrounding traffic so that they can make better driving decisions. [14] presents the concept of cooperative vehicle safety (CVS) which is aimed at achieving real-time situation awareness for the purpose of safer and possibly autonomous driving through vehicle cooperation. They propose a CPS approach to design a CVS system in which they model the cyber component (vehicle state delivery, networking) and the physical process estimation component in a unified framework. In a similar way, [15] discusses the concept of intelligent transportation spaces which is aimed at improving the vehicle and transportation safety, efficiency and sustainability. Intelligent transport spaces integrate not only cyber modules (communication and computational modules, satellites), but also physical modules (pedestrians, vehicles, roadside infrastructures, traffic management centers). [16] presents vehicular CPS in the context of scheduling problems which address human factors such as drivers' perceptions and reactions. They argue that a driver may not be able to receive more than one service (*e.g.*, parking navigation, traffic signals, safety warning message etc.) in a short period of time, even if the CPS can provide multiple services. They study the problem of human factor-aware service scheduling, in which the goal is to deliver at most n services, each having a time-dependent utility to a subset of intended drivers so as to minimize the system-wide total utility loss due to unsuccessful delivery of some services.

2.4.2 Intelligent Water Distribution Network

CPS based approaches to design intelligent water distribution networks have been proposed with the goal of providing potable water to the consumers. [17] proposes one such ap-

proach in which physical components (*e.g.*, valves, pipes, and reservoirs) are coupled with the hardware and software to supports intelligent water distribution to consumers. Sensors distributed in the physical environment collect information such as consumer demand patterns, water flow pressure, and water quality, and send this information to the algorithms running on the cyber-infrastructure. These algorithms provide decision support to the hardware controllers that manage quantity and quality of the water.

2.4.3 Smart Power Grid

Smart grids are power distribution networks that use a cyber-infrastructure to improve the efficiency and reliability of production and distribution of electricity by gathering and acting on various types of information such as consumers' demand pattern and suppliers' production patterns. [18] introduces a model of a cyber-physical energy system that views such a system as the interconnected cyber-physical network of many non-uniform components, such as diverse energy sources, and different types of energy users who may be equipped with their own local energy sources. Their proposed approach is based on representing all physical components as modules interconnected by means of an electric network. All physical components and/or groups of components are monitored using extensive signal processing methods and sensing. This model provides support to adequately decide what to sense and at which rate, what level of data mining is needed for which (groups of) physical components to achieve predictable performance for power grid.

In a similar fashion, [19] also views a smart power grid as a real-time system having both cyber and physical components. However, it focuses on model checking aspects of CPS. The authors claim that if operated independently, each component of smart grid may function correctly, but may fail to function correctly when integrated. This may be due to a specific type of interference in the frequency domain, which violates the Nyquist rate. They propose various techniques to verify the correctness of the cyber-physical composition using model-checking. The main challenge is to encode signal processing problem characteristics

into a form that can be model checked.

2.4.4 Autonomous Garden

[20] proposes a distributed autonomous gardening system, which they model as a mesh network of robots and plants. The gardening robots are equipped with an eye-in-hand camera, and are capable of locating plants, watering them, and locating and grasping fruits. The plants are equipped with sensors to monitor their health (*e.g.*, soil quality, state of fruits). They also have communication capabilities to make servicing requests to the robots. Task allocation to the robot and coordination among robots are implemented using distributed algorithms.

The models discussed above have paved a way towards solving interesting CPS problems in various multidisciplinary domains. Some of them have addressed challenges presented in Section 2.2; however they have not attempted to solve general problems such as mutual exclusion, and predicate and global state detection problems. Some work has been done towards solving predicate and global state detection for pervasive systems and CPSs, but they have their own shortcomings which we will discuss in following chapters.

Chapter 3

CPS Model

Model-driven development [21], and model-based design [22] have been studied extensively in the past, in which model plays a pivotal role in the system design process. A system model specifies all aspects of the system that can be modeled under consideration, and enables the system developers to analyse and simulate the system, so that design flaws and errors can be identified early in system development life cycle. CPSs, as we have discussed, are complex in nature due to involvement of large number of heterogenous components and physical processes, and therefore, they can benefit from this modeling paradigm.

A CPS model should facilitate the designer by providing tools to model the physical and cyber infrastructures, interactions between physical and cyber entities, behavior of the physical entities in the physical infrastructure, and the algorithms running in the cyber infrastructure. In this chapter, we propose a model of CPS, which enables the designers to model all the aspects of CPSs which we have discussed above. To ease the mathematical complexities inherent in the model, we propose a *Cyber-Physical System Modeling Language* (CPSML) , which enables the designer to capture the CPS model in easy to understand programming language. Throughout the chapter, we use a hospital CPS as an example to explain various constructs of our proposed model and CPSML.

3.1 Related Work

Increasing interest of research community in the field of CPS has driven researchers to propose CPS models providing different capabilities. We discuss two CPS models which are close to our proposed model (discuss in the next section).

Taha and Philippsen [23] recently proposed a small, experimental language to model CPSs. This language provides a number of constructs as discussed below:

- Ground values, such as $true$, $false$, string literals, integer and real constants.
- Vectors and matrices, such as $\begin{pmatrix} 5 & 8 \end{pmatrix}$ and $\begin{pmatrix} 1 & 4 \\ 3 & 7 \end{pmatrix}$.
- Object denition, such as defining classes.
- Object instantiation and termination.
- Variable declarations, including a special variable called _3D for generating visualizations
- Variable derivatives
- Continuous assignment
- Discrete assignment
- Conditional statements (if, and switch)
- Expressions and operators on reals

This model provides a unique feature that aids in visualization (_3D variables) and animation (continuous assignment). For example, Listing 3.1 specifies a particular way for drawing a robot. Parameter m represents a mass, which is used to specify a size and a color for the robot. Parameter D is a display reference point, which specifies the place where the robot needs to appear.

```
class robot (m,D)
    private
        p =[0,0,1];
        _3D = [["Robot", D+[0,0,1],
        0.03*sqrt(m),
        [m/3,2+sin(m),2-m/2],
        [1,1,1]]];
    end
    _3D [=] [["Robot", D+p,
    0.03*sqrt(m),
    [m/3,2+sin(m),2-m/2],
    [1,1,1]]];
end
```

Listing 3.1: *A class specifying definition of a robot*

In order to create an animated robot, one first creates a robot by writing `s = create robot(m,D)` in the initialization section, and then by letting the value of p varying over time by using continuous assignment [=] operator (see Listing 3.2).

```
class moving_robot (m,D)
    private s = create robot (m,D);
        t = 0;
        t1 = 0;
    end
    t1  [=] 5;
    s.p [=] [sin(t)*sqrt(1-(sin(t/10)^2)),
    cos(t)*sqrt(1-(sin(t/10)^2)),
    sin(t/10)];
end
```

Listing 3.2: *A class specifying an animated robot*

Even though this language provides special constructs to aid in visualization and anima-

tion, it lacks capabilities to capture user behavior, interaction, and physical infrastructure, which are essential elements of CPSs.

Kshemkalyani et al [24] defined a model of pervasive systems at the application layer, which divides a CPS into physical system and cyber systems. This model defines the system as a quadruple $\langle P, L, O, C \rangle$ where:

- P is a set of sensor/actuator processes which have access to some form of clock,
- L is a network of processes in P which can communicate with each other via asynchronous message-passing,
- O is a set of physical world objects, each having a set of attributes, that can be sensed and/or controlled by the processes in P, and
- C is a network in the physical world over which the objects in O communicate (in a synchronous or asynchronous manner).

$\langle P, L \rangle$ forms the network plane. The processes in P may be static or mobile and may communicate over wired or wireless media with one another over L. A process in P can also sense and actuate the objects in its range. $\langle O, C \rangle$ forms the physical plane. The objects in O (such as nurses, doctors, animals) may be static or mobile. These objects can be sensed by and/or can receive actuator signals from processes in P, but have no independent access to a synchronized clock. The objects in O can communicate with one another over C. While this model captures some of the aspects in our model, it does not define a spatial model of the physical system and behavior of physical objects, which is crucial in the type of applications we want to address.

To overcome these shortcomings, we have proposed a graph based CPS model [25] to specify CPSs. There are two parts to the specification of a CPS. The first is the specification of the CPS infrastructure, and second is the specification of the behavior of the CPS users. In the following sections, we present a CPS model and CPSML.

3.2 Specification of CPS infrastructure

We specify the infrastructure of a CPS as a triple $(CyS, PhyS, Int)$, where CyS represents the cyber infrastructure or the cyber-subsystem, $PhyS$ represents the physical infrastructure or the physical-subsystem and Int represents the interaction between CyS and $PhyS$. In the following, we explain each component of the CPS triple.

3.2.1 CyS: The Cyber-Subsystem

The cyber-subsystem models the computing elements of the CPS and is defined in the same way as a TDS, *i.e.*, using a graph $G_C = (CE, E)$, where CE is a set of cyber entities (computing platforms) and E is a set of edges. An edge $E_{ij} \in E$ between two cyber entities C_i and C_j represents a communication link between C_i and C_j. Each $C \in CE$ has a set of processes, denoted by $V.processes$, running on it. Processes executing on cyber entities communicate via the communication links to interact with each other. This model is shown in Figure 1.1(a)

3.2.2 PhyS: The Physical-Subsystem

The physical-subsystem models the physical elements (spatial properties, users, and resources) of the CPS. It is defined as a pair (PE, G_P), where PE is a set of physical entities and G_P represents the spatial model of the CPS. Depending on the application requirements, G_P can represent a $flat$ or an *hierarchical* spatial model, which we discuss in the rest of the section.

Flat Spatial Model

In the flat spatial model, G_P is represented as a graph (PA, RE), where PA is a set of physical areas and RE is the set of reachability edges. The concepts of *physical areas* and *reachability edges*, are defined below.

Definition 3.1. *(Physical area): We define a physical point pp in a 2D space as* $pp \in \mathbb{R}X\mathbb{R}$ *and a physical area A as set of* 2 *or more physical points.*

A physical area could represent a circle (two points: center, and one point on its circumference), or it could represent a set of the end points of a bounded polygon (three or more points). Each physical area is given a name, for example, the physical area constructed by joining four points (shown in green color) in Figure 1.1(b) is given a name A_1. Similar to Chandran and Joshi's assumption [11], we assume that a location naming service exists which maps distinct names (*e.g.*, $Room1$, $ICU1$) to underlying geometric representation by generating the geometric details of the physical areas.

Definition 3.2. *(Reachability Edge): If a physical entity can move directly between two physical areas* A_i *and* A_j*, we say a reachability edge* R_{ij} *exists between those areas.*

For example, in Figure 1.1(b), since there is a doorway connecting A_1 and A_4, there is a reachability edge between them. The notion of reachability edges which is missing in existing spatial models has following advantages:

- If all physical areas are represented as circles with same radii, then one can calculate approximate distance between any two physical areas.
- If reachability edges are associated with some properties, then one can search for property specific paths. For example, one can calculate path between two physical areas such that there is no staircase involved in the path.
- It provides a way to predict future movements of a physical entity. For example, if a nurse N_1 is located in A_{15} (see Figure 1.2), then the nurse can only move to A_{16} or A_3. It also helps in predicting a possible location of physical entities. For example, if N_1 moves out of A_{15} and is currently not in the sensing range of sensors, then one can predict that the nurse may be in an intermediate area between A_{15} and one of A_3, or A_{16}.

The flat spatial model does not distinguish between logical and physical areas, and fine and coarse grained areas. These features are present in Chandran and Joshi's spatial model [11], but it lacks the concept of reachability edges. In the following, we propose an hierarchical spatial model which combines the features of [11] and reachability edge notion of [25].

Hierarchical Spatial Model

In the hierarchical spatial model, G_P is represented as a *physical area tree* (PAT), which is a rooted tree structure of physical areas. It allows us to maintain hierarchical location information such as the following: if a patient is present in ICU, then the patient is also present in $Hospital$. In order to define PAT, we need to define additional concepts such as *logical area*, *contains relation*, *fine grained area*, *coarse grained area*, *parent area*, and *child area*.

Definition 3.3. *(Logical area): A logical area represents a type which characterizes a set of physical areas which exhibit similar properties.*

In other words, a logical area represents a set whose elements are physical areas. The concept of the logical area has been studied extensively [11, 26, 27]. They discuss it in the form of "spatial feature type", "logical location", and "spatial realms" respectively. In Figure 1.2, $Room$ is an example of a logical area and $Room1$ and $Room2$ are of type $Room$. Thus, $Room = \{Room1, Room2\}$, or we can also say that, $Room1$ and $Room2$ are instances of $Room$. For each physical area A, we define the set of physical points exterior to the physical area as A^e, interior to the physical area as A^i and on the boundary of the physical area as A^b, similar to the convention used in [11, 28].

Definition 3.4. *(Contains Relation $\mathbb{C}$): For two physical areas A_1 and A_2, $A_1 \mathbb{C} A_2$ if $(A_2^i \subset A_1^i) \wedge (A_2^b \subset A_1^i)$.*

Lemma 1. *($\mathbb{C}$ is transitive): if $A_i \mathbb{C} A_j$ and $A_j \mathbb{C} A_k$ then $A_i \mathbb{C} A_k$.*

Proof. From definition of $\mathbb{C}$, we get

1. $(A_j^i \subset A_i^i) \wedge (A_j^b \subset A_i^i)$
2. $(A_k^i \subset A_j^i) \wedge (A_k^b \subset A_j^i)$

From 1 and 2 above, we get $(A_k^i \subset A_i^i)$ and $(A_k^b \subset A_i^i)$, which implies that $A_i \mathbb{C} A_k$. □

Definition 3.5. *(Overlap Relation $\mathbb{O}$): A physical area A_1 is said to overlap another physical area A_2 if $(A_2^i \cap A_1^i \neq \emptyset) \wedge (A_2^b \cap A_1^b \neq \emptyset) \wedge (A_2^e \neq A_1^e)$.*

Definition 3.6. *(Fine grained area): A physical area A_i is called a fine grained area if there does not exist A_j such that $A_i \mathbb{C} A_j$. All fine grained areas are instances of a special logical area of type "FINE".*

Definition 3.7. *(Coarse grained area): A physical area A_i is called a coarse grained area if there exists A_j such that $A_i \mathbb{C} A_j$. All coarse grained areas are instances of a logical area of some type.*

Definition 3.8. *(Parent and Child Area): A physical area A_j is called a parent area of another physical area A_i if $A_j \mathbb{C} A_i$ and the following two properties are satisfied:*

1. *there does not exists A_k such that $A_j \mathbb{C} A_k$ and $A_k \mathbb{C} A_i$, i.e., parent areas are not transitive.*
2. *there does not exist A_k such that $A_k \mathbb{C} A_i$ and property 1 is also satisfied (i.e., A_j does not contain A_k). This eliminates the possibility of an area having two parents.*

If A_j is parent area of A_i, then A_i is called a child area of A_j.

Definition 3.9. *(Root area): A coarse grained area is called the root area if it doesn't have any parent, and contains all other coarse and fine grained areas in a physical system.*

We illustrate the concepts we just defined using an example. In Figure 1.2, $Hospital1$, $Floor1$, $Room1$, $Room2$, $ICU1$, $NurseStation1$ are coarse grained areas, because each of

them contain another coarse or fine grained areas. Because $Hospital1$ does not have any parent, it is the root area. A_1 to A_{16} are fine grained areas, as they do not contain any other fine or coarse grained areas. Some of the examples of *contains* relation in Figure 1.2 are $Hospital1\mathbb{C}Floor1$, $Floor1\mathbb{C}ICU1$, $Floor1\mathbb{C}NurseStation1$, $ICU1\mathbb{C}A_{15}$, and $ICU1\mathbb{C}A_{16}$. From Lemma 1, $Hospital1\mathbb{C}ICU1$, $Hospital1\mathbb{C}A_{15}$, $Hospital1\mathbb{C}A_{16}$, $Floor1\mathbb{C}A_{15}$, and $Floor1\mathbb{C}A_{16}$. From Definition 3.8, $Hospital1$ is parent of $Floor1$, $Floor1$ is parent of $ICU1$, and $ICU1$ is parent of A_{15} and A_{16}, however, $Hospital1$ is not parent of $ICU1$.

The *contains* relation is useful in representing hierarchical information about physical entities. For example, if a user $U1$ is located inside area A_{15}, and since $ICU1\mathbb{C}A_{15}$, $Floor1\mathbb{C}A_{15}$, and $Hospital1\mathbb{C}A_{15}$, $U1$ is also said to be located in $ICU1$, $Floor1$, and $Hospital1$. We define PAT in the following section.

PAT

PAT is constructed by considering all fine and coarse grained areas as nodes and drawing an edge from a node A_i to a node A_j if A_i is parent of A_j. The parent-child relationship ensures that all physical areas in PAT except *Root* are have exactly one parent. Each non-leaf node in PAT represents a coarse grained area $Name$ which is an instance of logical area $Type$, and is represented as a tuple $\langle Name, Type\rangle$. Each leaf node represents a fine grained area, and is represented as a tuple $\langle A_i, \{R_{ij}\}\rangle$. In the tuple $\langle A_i, \{R_{ij}\}\rangle$, A_i is the fine grained area being monitored by cyber entity C_i, and $\{R_{ij}\}$ is the set of fine grained areas A_j such that R_{ij} exists. We assume that the sensing range of cyber entities any two cyber entities C_i, and C_j monitoring fine grained areas A_i and A_j respectively, do not overlap, *i.e.*, $A_i\mathbb{O}A_j$ does not hold. PAT for Figure 1.2 is shown in Figure 3.1. In the figure, *WA* stands for *Waiting Room*, *RR* stands for *Rest Room*, and *NS* stands for *Nurse Station*.

Operations on PAT

We allow the following operations on a PAT:

$\langle Hospital1, Hospital\rangle$
- $\langle Floor1, Floor\rangle$
  - $\langle A_4, \{A_1, A_2, A_5, A_8\}\rangle$
  - $\langle A_5, \{A_3, A_4, A_{12}, A_{13}\}\rangle$
  - $\langle A_8, \{A_4, A_6, A_7\}\rangle$
  - $\langle A_9, \{A_1, A_{10}\}\rangle$
  - $\langle A_{10}, \{A_9\}\rangle$
  - $\langle WA1, WA\rangle$
    - $\langle A_1, \{A_2, A_4, A_9\}\rangle$
  - $\langle RR1, RR\rangle$
    - $\langle A_7, \{A_8\}\rangle$
  - $\langle RR2, RR\rangle$
    - $\langle A_6, \{A_8\}\rangle$
  - $\langle Room1, Room\rangle$
    - $\langle RR3, RR\rangle$
      - $\langle A_{11}, \{A_{12}\}\rangle$
    - $\langle A_{12}, \{A_5, A_{11}\}\rangle$
  - $\langle Room2, Room\rangle$
    - $\langle RR4, RR\rangle$
      - $\langle A_{14}, \{A_{13}\}\rangle$
    - $\langle A_{13}, \{A_5, A_{14}\}\rangle$
  - $\langle ICU1, ICU\rangle$
    - $\langle A_{15}, \{A_3, A_{16}\}\rangle$
    - $\langle A_{16}, \{A_{15}\}\rangle$
  - $\langle NS1, NS\rangle$
    - $\langle A_2, \{A_1, A_3, A_4\}\rangle$
    - $\langle A_3, \{A_2, A_5, A_{15}\}\rangle$
  - $\vdots$
- $\langle OtherFloors, Floor\rangle$

Figure 3.1: *PAT for Figure 1.2*

- **Root**: It returns the root area of PAT.

- **A.Type**: It returns the set of all physical areas which are instances of logical area $Type$ and are contained in coarse grained area A. Note that $A.FINE$ will return the set $\{A_i | A_i \text{ is a fine grained area} : A \mathbb{C} A_i\}$

- **A.Contained**: It returns the set of all physical areas A_i such that $A_i \mathbb{C} A$. For example, in Figure 1.2, $A_{15}.contained$ returns the set $\{ICU1, Floor1, Hospital1\}$

- **A.Entrance**: For a coarse grained area A, $A.Entrance = \{A_i | A \mathbb{C} A_i \wedge (\exists A_j : \exists R_{ij} \wedge (A, A_j) \notin \mathbb{C})\}$

- **A.Exterior**: For a coarse grained area A, $A.Exterior = \{A_i | (\exists A_j \in A.Entrance : \exists R_{ij}) \wedge (A, A_i) \notin \mathbb{C}\}$

- **A.Interior**: For a coarse grained area A, $A.Interior = \{A_i | A \mathbb{C} A_i \wedge A_i \notin A.Entrance\}$

  For example, Figure 3.2 depicts a coarse grained area $ICU2$ of type ICU. The circle with a letter i indicates a fine grained area A_i contained in $ICU2$ and sensed by a cyber entity C_i. The circle also indicates the sensing range of C_i. A Curve between two fine grained areas represents a reachability edge between them. In this figure, $ICU2.Entrance = \{A_6\}$, $ICU2.Exterior = \{A_7, A_8\}$ and $ICU2.Interior = \{A_1, A_2, A_3, A_4, A_5\}$.

- **Path($\mathbf{A_s}$, $\mathbf{A_d}$)**: For two fine grained areas A_s and A_d, it returns a sequence $\mathcal{S}$ of unique fine grained areas $\{A_1, \ldots, A_n\}$ such that $A_s, A_d \notin \mathcal{S}$, and $R_{s1}, R_{12}, \ldots, R_{(n-1)n}, R_{nd}$ exist. If number of elements in the sequence is n, then the length of the path is $n+1$, *i.e.*, it requires a physical entity to travel $n+1$ reachability edges to move from A_s to A_d.

  Note that either one or both of A_s and A_d can be coarse grained areas. If both are coarse grained areas, then $Path(A_s, A_d)$ returns a sequence of unique fine grained areas

$\{A_1, \ldots, A_n\}$ such that $R_{12}, \ldots, R_{(n-1)n}$ exist, $A_s \mathbb{C} A_1$, and $A_d \mathbb{C} A_n$. In this case, the path length is $n-1$. We can similarly define this operation when only one of A_s and A_d is a coarse grained area.

- **ShortestPath**($\mathbf{A_s}, \mathbf{A_d}$): It returns a path of length n between A_s and A_d such that there is no other path between A_s and A_d whose length is less than n.

- **Hop**($\mathbf{A_1}, \mathbf{A_2}$): It returns the length of the shortest path between A_1 and A_2.

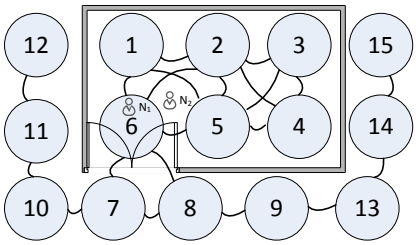


Figure 3.2: *Part of a hospital showing an intensive care unit*

PE: The set of Physical Entities

Physical entities represent physical elements of a CPS. For example, hospital staff, medical equipments, and patients are physical entities in a hospital CPS. We divide the physical entities into two categories: active entities and resources. Active entities are the users of the system and can perform actions on their own (*e.g.*, hospital staff) and may use the resources (*e.g.*, wheelchairs). Each concrete instance of active entity and resource has a type which we define as follows:

Definition 3.10. *(Active Entity Type): It is a type which characterizes a set of active entities which exhibit similar properties.*

Definition 3.11. *(Active Entity Instance): An active entity instance is an instance of active entity type.*

Definition 3.12. *(Resource Type): It is a type which characterizes a set of resources which exhibit similar properties.*

Definition 3.13. *(Resource Instance): A resource instance is an instance of a resource type.*

For example, $Nurse$ is an active entity type and N_1 is an instance of type $Nurse$. Similarly, $WheelChair$ is a resource type and WC_1 is an instance of type $WheelChair$. PE is defined in terms of above definitions as a tuple (AE, RS) where:

- AE is defined as a tuple $(\mathcal{T}_{ae}, \mathcal{I}_{ae})$ where $\mathcal{T}_{ae}$ is the set of all active entity types which we allow to be present in *PhyS*, and $\mathcal{I}_{ae}$ is set of all active entity instances present in *PhyS*, each of them being an instance of exactly one of the active entity types in $\mathcal{T}_{ae}$. Each $\tau_{ae} \in \mathcal{T}_{ae}$ has a set of properties $\tau_{ae}.prop = \{prop_1, \ldots, prop_n\}$ associated with it which is also instantiated for its instances. For an active entity instance ι_{ae} of τ_{ae}, $\iota_{ae}.prop_i$ returns the value of $prop_i$ of ι_{ae}. For example, if $Nurse.prop = \{name, age\}$, the elements of $N_1.prop$ will have some values associated with them and $N_1.name$ and $N_1.age$ return those values. For an ι_{ae}, we use the expression $\iota_{ae}.type$ to return its type and $\iota_{ae}.location$ to return the fine grained area in which it is located.

- RS is defined similarly as a tuple $(\mathcal{T}_r, \mathcal{I}_r)$ where $\mathcal{T}_r$ is the set of all resource types which we allow to be present in *PhyS*, and $\mathcal{I}_r$ is set of all resource instances present in *PhyS*, each of them belonging to exactly one of the resource types in $\mathcal{T}_r$. Similar to active entity types and instances, resource types and instances also have a set of properties associated with them. One of the mandatory properties of resource types is *state* which is either $free$ or *busy*. For resource instance ι_r, we use the expression $\iota_r.type$ to return its type and $\iota_r.location$ to return the fine grained area in which it is located.

All active entities and resources are located in some physical (fine and coarse grained) area, which we model as follows.

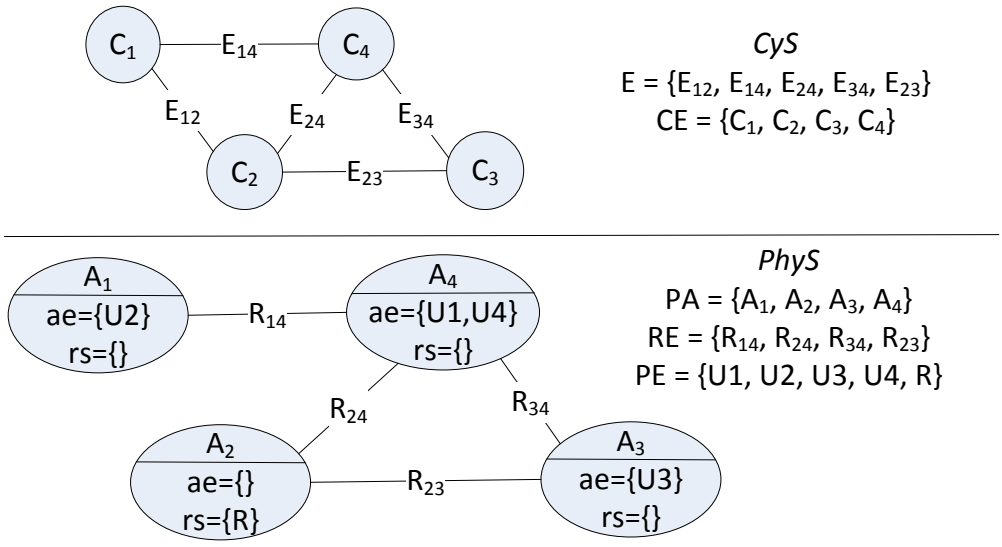


Figure 3.3: *Detailed view of* CyS *and* PhyS *of* CPS *shown in Figure 1.1(b)*

Abstract Representation of a Physical Area

We model each physical area A in flat spatial model (or fine grained area in hierarchical spatial model) by two abstract variables, $A.ae$ and $A.rs$, which denote the set of active entity instances and set of resource instances respectively currently located in area A. Figure 3.3 shows these values for the CPS shown in Figure 1.1(b). We assume that these abstract variables are updated automatically based on the actions (defined later in this section) of the active entities. For instance, if $U \in A.ae$, and U moves out of the area A, then the state variable $A.ae$ is automatically updated so that U is removed from $A.ae$. Similarly, when U enters a new area, the corresponding state variable is updated to include U. We assume a similar update happens for $A.rs$ when resources are moved between areas.

In an hierarchical model, if A is a coarse grained area, then $A.ae = \bigcup_i A_i.ae$ such that $A \mathbb{C} A_i$. Similarly, $A.rs = \bigcup_i A_i.rs$ such that $A \mathbb{C} A_i$.

For a physical area A, active entity type τ_{ae}, and resource type τ_r, $A.\tau_{ae}$ represents the set of active entity instances of type τ_{ae} located in area A, and $A.\tau_r$ represents the set of resource instances of type τ_r located in area A

3.2.3 Int: Interaction between CyS and PhyS

Int captures the capabilities for interaction between *CyS* and *PhyS*. Interactions are defined in terms of actions and events as $action \rightarrow event$. It implies one of following two:

1. *action* performed by an active entity in *PhyS* causes *event* to be generated in any of the cyber entities in *CyS*, or

2. *action* performed by a cyber entity in *CyS* causes *event* to be generated at any of the active entities in *PhyS*. The only possible scenario for this type of interaction is when a cyber entity sends a message to an active entity U (assuming that active entities carry a hand held device to digitally communicate with *CyS*). In other words, a send message action performed by the cyber entity causes a receive message event to occur at the active entity.

Assume that an area A is monitored by a cyber entity C. In order for actions performed by active entities in A to cause events to be generated in cyber entity C, we allow C to sense the presence of physical entities in A. We model this by allowing processes in $C.processes$ to read $A.ae$ and $A.rs$, including their properties, *i.e.*, each C maintains its own set of variables $A.ae$ and $A.rs$. These variables change as the state of A changes due to various actions performed by active entities.

Note that because of sensing and communication delays, generation of *event* is asynchronous with occurrence of *action*. *i.e.*, if *action* is performed at time t, then corresponding event is generated at time $t + n$, where n is less than or equal to sensing delay.

$\boldsymbol{Move(A) \rightarrow Sense(U)}$ and $\boldsymbol{Move(A) \rightarrow Lose(U)}$

$Move(A)$ is an action which represents an active entity U moving from its current physical area to another physical area A, and is possible only if there exists a reachability edge from its current physical area to A. This action causes one of the following two events to occur in a cyber entity in *CyS*.

1. $Sense(U)$: When U moves into an area A monitored by a cyber entity C, and C starts sensing the presence of U, we say a $Sense(U)$ event has been generated in C. As a consequence of this event, C includes U into its variable $A.ae$.

2. $Lose(U)$: When U moves out of an area A monitored by a cyber entity C, and C stops sensing the presence of U, we say a $Lose(U)$ event has been generated in C. As a consequence of this event, C removes U from its variable $A.ae$.

$Acquire(rs) \rightarrow ChangeState(rs)$ and $Release(rs) \rightarrow ChangeState(rs)$

The action $Acquire(rs)$ represents the attempt by an active entity U to physically acquire a resource rs which results in $rs.state$ being set to *busy*. There could be situations where several active entities may attempt to acquire the same resource (*e.g.*, if several people attempt to grab a wheelchair simultaneously, only one of them will be successful). To model this, we assume that the *Acquire* action is successful only if $rs.state$ is $free$, and that a successful *Acquire* will automatically change $rs.state$ to *busy*. Thus, $Acquire(rs) :< rs.state = free \rightarrow rs.state = busy >$ by U can be viewed as an atomic action which returns *true* if U successfully acquires rs; else it returns $false$.

$Release(rs)$ is used to physically release a resource rs which results in $rs.state$ being set to $free$, and is successful only if $rs.state$ is *busy*. Thus, $Release(rs) :< rs.state = busy \rightarrow rs.state = free >$ is an atomic action.

Both $Acquire(rs)$ and $Release(rs)$ actions cause $ChangeState(rs)$ event to be generated in a cyber entity C which is sensing the presence of rs. As a consequence of this event, C changes $rs.state$ to *true* or $false$, depending on whether $Release(rs)$ or $Acquire(rs)$ action is performed.

$Observe \rightarrow NULL$

$Observe()$ is an action which represents an active entity U observing physical entities within an *observation radius* (O_R). If O_R of U located in area A is 1, then U can observe the status

of A, *i.e.*, U can read $A.ae$ and $A.rs$, and $r.state$ for each resource $r \in A.rs$. In general, if O_R of U is r, then U can observe the status of all areas reachable via at most r hops in G_P. For implementation purposes, *Observe()* returns the set of resources which U can observe depending on its O_R.

Observe() action does not cause any event to occur in *CyS*. It is internal to the active entity and based on this action, the active entity performs other actions. For example, if an active entity U observes a free resource rs, then U can perform $Acquire(rs)$.

$\boldsymbol{Send(Msg) \rightarrow Receive(Msg)}$

While $Move$ and *Observe* actions can help U to solve a problem on its own (for example to locate a resource), it can also request *CyS* to solve the problem. U can send a request by performing the action $Send(Msg)$. When a cyber entity C wants to send some information to an active entity U, or want to communicate with another cyber entity, C performs $Send(Msg)$ action.

As a result of $Send(Msg)$ action performed by an active entity, a $Receive(Msg)$ event is generated in one or more cyber entities, and if performed by a cyber entity, a $Receive(Msg)$ action is generated in an active entity, or one or more cyber entities.

Msg is modeled as as a quadruple $\langle id, dest, type, payload \rangle$, where id is message identification number, *dest* is the destination of the message, *type* is message type, and *payload* is the actual message. id is a long integer which uniquely identifies the message in the system. When an active entity U performs $Send(Msg)$, the destination is either one of the cyber entities, or it can be *BCAST*. A *BCAST* in this case means that all the cyber entities which are in communication range of U will receive this message. Similarly, when a cyber entity C_i performs $Send(Msg)$, the destination is either an active entity, or another cyber entity, or it can be *BCAST*. *BCAST* in this case means that all cyber entities C_j such that E_{ij} exists will receive this message. We assume that each cyber entity has a routing table so that the message can be forwarded to the destination. Similarly, the hand-held device has

the capabilities to forward the message to the destination.

3.3 Specification of User's Behavior

A user's behavior is a sequence of valid actions performed by the user or the events generated at the user. For example, the following sequence may allow a user U to find a free resource. $Observe$, $Send(Msg)$, $Receive(Info)$, $Move$, $Move$, $Observe$, $Acquire(rs)$. We interpret this sequence as follows: U observes the area in which it is located. If the user does not find any free resource, and it requests CyS to locate a free resource and subsequently receives path to a free resource. then U follows the path received from CyS to move to that location (sequence of two moves). On reaching the destination area, it observes and acquires a free resource. Consider another example in which temporal notation is involved. A nurse N performs the following sequence of actions: $Move$ at 10:00 AM, $Observe$, $Move$ at t, $Move$ at $t + 5$. This sequence of actions can be interpreted as follows: N enters ICU at 10:00 AM. The nurse observes that there are no patients and leaves ICU at time t, and then comes back to the ICU after 5 minutes. Note that the nurse evaluates a condition *are there any patient in the ICU*, and depending on the outcome of the condition, the nurse takes further action. There may be scenarios in which the nurse needs to perform certain actions repetitively until certain condition is met. For example, the nurse may need to check a patient's vital signs every half an hour for the next 5 hours.

From the examples above, we find that the users' behavior consists of sequence of actions, events, and decision making depending on $Observe()$ action, and can be represented as a program (see Listing 3.3 and Listing 3.4). A user operates inside a physical system; therefore, given the specification of the physical system, we can write users' behavior. Moreover, we can also specify cyber entities' algorithms to complete the specification of a CPS. In order to achieve this, we propose *Cyber-Physical System Modeling Language* (CPSML), a domain specific language which allows us to model infrastructure (CyS and $PhyS$), users' behavior

and cyber algorithms. The grammar of the language is given in Appendix A. CPSML is a tool to specify CPS- it is *not* a language meant to generate executable code. A brief overview of the constructs of CPSML is presented in the next section.

```
User Behavior:
    rs = Observe(),
    Send("Locate a free wheelchair"),
    Receive(PATH),
    Repeat
      Move(A),
        remove A from PATH,
    until (PATH is null)
    rs = Observe(),
    if rs1 is a free resource in the set rs then
        Acquire(rs1),

Cyber Algorithm:
    while(true)
        Msg = receive(),
        Search free wheelchair and calculate the path,
        Send(path),
```

Listing 3.3: *Sequence of actions and events when a user requests* CyS *to locate a wheelchair*

```
Nurse Behavior:
    // A4 is in ICU.entrance and A3 is in ICU.exterior
    if(time == 10:00AM) then
        Move(A4),
    ae = Observe(),
    if(ae is empty) then
        time t = current_time,
        Move(A3),
    if(time == t+5) then
```

```
                Move(A4),
```

Listing 3.4: *Sequence of actions and events when a nurse moves in and out of ICU*

3.4 CPSML Overview

CPSML is a language which provides constructs to specify a CPS model in a similar way an algorithm is specified using well formed constructs. A CPSML program consists of following 6 sections:

Declaration Section: This section declares the types and instances of active entities, resources, and physical areas. Assume that in Figure 1.2, in addition to the nurses N_1, N_2, and N_6, three patients, one doctor two wheelchairs, and 2 X-Ray carts are also located in the system. Also assume that nurses, doctors, and patients have two properties: 'name' and 'ID'; and wheelchairs and X-Ray carts have 'ID' as the only property. This is represented in CPSML as shown in Listing 3.5

```
AE Declaration Begin
  Nurse(name, ID): N1(Wendy, N12), N2(Rowdy, N23), N6(Lisa, N33);
  Doctor(name, ID): D1(John, D22);
  Patient(name, ID): P1(Jay, P223), P2(Shawn, P554), P3(Stuart, P5676);
AE Declaration End

RS Declaration Begin
  WheelChair(ID) : WC1(WC1), WC2(WC2);
  XRay(ID): X1(X1), X2(X2);
RS Declaration End
```

Listing 3.5: *Active entity and resource declaration*

In our model, we specify physical infrastructure in one of the following two ways: flat and hierarchical. A flat physical model does not have a notion of logical areas, therefore,

in CPSML, we do not declare physical areas in this section, however, we define G_P in *Physical Infrastructure Specification Section*. For hierarchical physical infrastructure, we declare types and instances of physical areas as shown in Listing 3.6.

```
Area Declaration Begin
  Hospital: Hospital1;
  ICU: ICU1;
  Floor: Floor1;
  NurseStation: NS1;
  RestRoom: RR1, RR2, RR3, RR4;
  Room: Room1, Room2;
  WaitingArea: WA1;
  Fine: A1, A2, A3, A4, A5, A6,
      A7, A8, A9, A10, A11, A12,
      A13, A14, A15, A16;
Area Declaration End
```

Listing 3.6: *Declaration of hierarchical physical infrastructure as shown in Figure 1.2*

Cyber Infrastructure Specification Section: This section specifies G_C, *i.e.*, the topology of *CyS*. Each cyber entity is listed as $ce < n > (comma_separated_list_of_processes)$, where n is cyber entity Id. Communication edge between cyber entities m and n is represented as $e < m >_< n >$. For example, Listing 3.7 shows 4 cyber entities, each having one process, and Listing 3.8 shows 16 cyber entities, each having one process.

```
CyS Begin
  CE = (ce1(p1), ce2(p2), ce3(p3), ce4(p4));
  E = (e1_2, e2_3, e1_4, e2_4, e3_4);
CyS End
```

Listing 3.7: *Cyber infrastructure specification corresponding to Figure 1.1(b)*

```
CyS Begin
  CE = (ce1(p1), ce2(p2), ce3(p3), ce4(p4), ce5(p5), ce6(p6),
```

```
    ce7(p7), ce8(p8), ce9(p9), ce10(p10), ce11(p11), ce12(p12),
    ce13(p13), ce14(p14), ce15(p15), ce16(p16));
  E = (e1_2, e1_4, e1_9, e1_10, e4_9, e4_8, e4_6, e4_2,
    e4_3, e4_5, e4_11, e4_12, e12_11, e12_5, e5_3, e5_13,
    e3_15, e13_14, e14_16, e15_16, e15_5);
CyS End
```

Listing 3.8: *Cyber infrastructure specification corresponding to Figure 1.2*

Physical Infrastructure Specification Section: This section defines G_P in flat infrastructure or PAT in hierarchical infrastructure. Examples are shown in Listing 3.9 and Listing 3.10, and are self explanatory.

```
PhyS Begin
  PA = (a1, a2, a3, a4);
  RE = (re1_4, re4_2, re2_3, re4_3);
PhyS End
```

Listing 3.9: *Physical infrastructure specification corresponding to Figure 1.1(b)*

```
PhyS Begin
  Hospital1 contains Floor1 ;
  Floor1 contains ICU1, RR1, RR2, Room1, Room2, NS1,
              WA1, A8, A9, A10, A4, A5 ;
  ICU1 contains A15, A16;
  Room1 contains A12, RR3;
  RR3 contains A11;
  Room2 contains A13, RR4;
  RR4 contains A14;
  RR1 contains A7;
  RR2 contains A6;
  ICU1 contains A15, A16;
  NS1 contains A2, A3;
  WA1 contains A1;
```

```
    RE = (re1_2, re1_9, re1_10, re1_4, re4_2, re4_3, re4_5
16       re4_8, re6_6, re8_7, re5_2, re5_3, re5_12, re5_13
         re12_11, re13_14, re3_2, re3_15, re15_16);
18  PhyS End
```

Listing 3.10: *Physical infrastructure specification corresponding to Figure 1.2*

Constraints Section: This section specifies constraints of underlying CPS, and is discussed in detail in Section 5.3.

Active Entity Behavior Section: In this section, we specify behavior of all active entities which are declared in *Declaration Section*. The specification is enclosed within *AE Behavior Begin* and *AE Behavior End* phrases. If two entities of the same type exhibit similar behavior, we combine them together instead of writing two separate behaviors. Example of active entity behavior specification is shown in Listing 3.12. Users $U1$ and $U3$ looks for wheelchair in all possible areas until they acquire one successfully. User $U3$ on the other hand looks for a wheelchair only in the area (s)he is located into. CPSML provides following constructs to specify interactions:

- acquire(resource_id)
- release(resource_id)
- message = receive()
- move(fine_grained_area_id)
- send(message)
- sense(active_entity_id/resource_id)
- lose(active_entity_id/resource_id)
- change_state(resource_id)

- list = observe()

The message is defined using ⟨*message*⟩, and an example is shown in Listing 3.11. This message represents a response to nurse N1's request to locate a free wheelchair.

⟨message⟩→
'{'
'ID' ⟨equal⟩ ⟨int⟩ ⟨comma⟩
'dest' ⟨equal⟩ ⟨id⟩
'type' ⟨equal⟩ ⟨id⟩
'payload' ⟨equal⟩ **(** ⟨character⟩ | ⟨digit⟩ **)***
'}'

```
Message msg1 = {
            id = 22343,
            dest = N1,
            type = RESPONSE,
            payload = "A1 A2 A3 A15"
          }
```

Listing 3.11: *Message structure.*

```
AE Behavior Begin
  User: (U1, U3);
  L1: pe[] = observe();
    resources[] = pe[].rs;
    if(sizeof(resources[]) != 0) {
      for (each r in resourcess[]) {
        if(r.state == free) {
          val = acquire(r);
          if(val == false){
```

```
            continue;
          }
        }
      }
    }
    n = select unvisited edge from RE;
    move(n);
    goto L1;

  User: (U2);
  L1: pe[] = observe();
    resources[] = pe[].rs;
    if(sizeof(resources[]) != 0) {
      for (each r in resourcess[]) {
        if(r.state == free) {
          val = acquire(r);
          if(val == false){
            continue;
          }
        }
      }
    }
AE Behavior End
```

Listing 3.12: *Active entity behavior specification.*

Cyber Algorithm Section: In this section, we specify the algorithms running in the processes in the cyber entities. We use Promela [29] like constructs, specifically timer and control flow structures, to specify cyber algorithms. [1]

Listing 3.13 shows a simple cyber algorithm specification section which defines a process

[1]Promela is a process modeling language used to model check distributed and concurrent systems. Refer to [30] for an up-to-date details of the language.

type `P`. The process is started using `run P(n)` where `n` is Id of the process. The specification starts two processes of type `P`, in which one process keeps sending a message to the other process.

```
CyS Algorithm Begin
    Message msg1 = {
                id = 0,
                dest = P2,
                type = REQUEST,
                payload = ""
            };
    Message msg2 = null;
    int i=0;
    proctype P(id)
    {
        do
        ::  (id == 1) --> i++; msg1.id = i; send(msg1);
        ::  (id == 2) --> msg2 = receive()
        od;
    };
    init
    {
        run P(1);
        run P(2);
    }
CyS Algorithm End
```

Listing 3.13: *Cyber algorithm specification.*

3.5 Summary

In this chapter, we looked at two existing CPS models and their shortcomings in the context of the problem which we are interested to solve. The model proposed in this chapter opens the possibility of studying more complex CPS models. These possibilities include:

- associating properties such as type, width, and length with the reachability edges in G_P. For example, type may include staircase or elevator. Width may indicate the maximum number of physical entities which can move via it at a given point time. The length represents the length of the reachability edge. In this chapter, we assumed that the length of all reachability edges are the same.

- extending spatial model to include multiple physical systems. For example, the spatial model may include a couple of hospitals.

Chapter 4

Mutual Exclusion

This chapter begins with an introduction of mutual exclusion in a TDS and continues to define mutual exclusion problem for a CPS with flat spatial model (Section 3.2.2). We propose both centralized and distributed solutions to the problem. We also discuss and compare the simulation results of our proposed algorithms.

4.1 Introduction

There are many systems in which users need exclusive access to shared physical resources. For example, as discussed in Section 1, users (*e.g.*, hospital staff) in a health care facility share physical resources (*e.g.*, wheelchairs, IV pumps) located in different parts of the facility. A traditional hospital may be using manual techniques to locate and share resources (*e.g.*, depositing free resources at a central or a set of known locations). Whereas in a smart hospital, the resources are instrumented with sensing devices to track their location and usage, and the information is made available to the potential users.

[31] describes a similar smart factory environment where context data (location and usage) regarding tools, machines, transport carts, and spare parts is made available via RFID tags to aid in locating the nearest tools/machines available to do a task. Similar

systems have been discussed in various contexts such as locating empty spaces in parking lots [32], room reservation in buildings [33] and smart building operations [34].

One central issue in many application scenarios such as discussed above is the use of resources in an exclusive manner. In this chapter, we study the problem of mutual exclusion in CPSs where users need exclusive access to physical resources. In a TDS, a mutual exclusion algorithm is typically modeled as a set of processes $P_1, \ldots, P_n$, where P_i executes on node C_i, and a strict layered structure is used wherein user Ui interacts with P_i to gain access to a resource. The access of the resources in a TDS is completely regulated by the mutual exclusion algorithm.

In a CPS, *CyS* has been superimposed on an existing physical system (Figure 1.2). In such a case, a distributed mutual exclusion algorithm executing in *CyS* must operate in the context of existing techniques being used to locate resources in *PhyS* (such combined use of cyber and physical techniques may indeed be more efficient as shown later by our experiments). In such cases, a distributed algorithm may have to contend with direct interactions between the users and the resources. This introduces several aspects (Section 2.2) in the context of the mutual exclusion problem which are not addressed in a TDS.

Several distributed computing problems such as distributed algorithms for creating global states in intelligent construction sites [35], event ordering [36, 37] and termination detection [38, 39] have been studied for CPSs. Although existing research discussed above has addressed some aspects of interactions between *CyS* and the users, the problem in the context of mutual exclusion, has not been addressed. Depending on the interactions between *CyS* and *PhyS*, there is a range of possible solutions for the mutual exclusion problem in a CPS. On one end, the users may ignore *CyS* and address the problem on their own by physically locating the resources (by actions of moving and observing). At the other extreme, one can follow the TDS approach wherein the users ask *CyS* to locate resources, and use them only as directed by *CyS*. In between these extremes, we can have an array of solutions depending on the cooperation between the users and *CyS*.

Our set proposed set of algorithms is based on CPS model with flat spatial model. As mentioned in Section 2.3, each algorithm has two components: one describing the behavior of the users in *PhyS*, and the other describing the mutual exclusion algorithm (or cyber algorithm) in *CyS*. Each combination of user behavior and cyber algorithm yields a different CPS algorithm. We have conducted an extensive simulation study of proposed algorithms using OMNeT++ [40] which simulates both user behavior and cyber algorithm. We have studied the impact of various factors such as the observation capabilities of the users, frequency of sensing, and behavior of the users on the time to acquire a resource.

4.2 Background and Related Work

A mutual exclusion algorithm for a TDS typically models a physical resource (*e.g.*, a printer) as an abstract object and provides users with an interface with functions to request, acquire and release a resource, and ensures that at most one user is granted access to a resource at a time. Mutual exclusion algorithms have been studied extensively [1–4] for both shared memory and message passing systems. In the more general k-mutual exclusion problem, at most k processes are allowed to be in critical section at the same time. Two main approaches have been studied to address the distributed k-mutual exclusion problem. In the token based approach [5–8], a process is allowed to enter the critical section only after the process has acquired a token. In the permission based approach [9], a process must request and be explicitly granted permission to enter the critical section from a specific subset of processes. While there has been significant research in distributed mutual exclusion algorithms, aspects specific to CPS such as users interacting with another and using additional mechanisms external to the algorithm to acquire and release resources, have not been studied.

In [8], a token based k-mutual exclusion algorithm, which is referred to as the KRL algorithm, was proposed for wireless ad hoc networks. As one of our solutions is based on the KRL algorithm, we discuss this algorithm in more detail in the following. In the KRL

algorithm, each node i maintains a data structure $height_i$, which is a three-tuple (h_1, h_2, i). Edges are directed from higher height nodes to lower height nodes based on lexicographic ordering. For example, if $height_0 = (2, 3, 0)$ and $height_1 = (2, 2, 1)$, then $height_0 > height_1$ and the edge will be directed from node 0 to node 1. *KRL* algorithm maintains n nodes and k tokens, where $k < n$. For all nodes i, $height_i$ is initialized so that the directed edges form a *directed acyclic graph* (DAG) such that every node has a directed path to some token holder and every token holder node i has at least one neighbor n such that $height_n > height_i$. When a user at node i wants to enter the critical section, it makes a request which is enqueued by P_i in its local queue Q_i. When P_i receives a request from neighbor P_j and $height_j > height_i$, P_i enqueues the request in Q_i. If P_i is a non-token holding node and Q_i is empty when the request is received, P_i sends a request to its neighbor with the lowest height. Hence, requests propagate via lower height nodes to the token holders. If P_i has (or receives) a token, it dequeues the first request from Q_i. If this request is from its own application process, P_i gives permission to its application process; else, it sends the token to its neighboring node whose request it just dequeued.

4.3 Mutual Exclusion in a CPS

In this section, we present mutual exclusion algorithms for CPS. Each algorithm has two components: (a) the behavior of active entities describing their efforts to locate resources and (b) a cyber algorithm.

4.3.1 Behavior of active entities

Behavior describes the steps followed by an active entity to locate a resource with the help of $Observe$, $Move$, and $Send(Msg)$ actions. We consider the following possible behaviors:

Behavior B_0: In this behavior, U searches for a resource without any help from *CyS*. At each step, U observes the area it is located into (Algorithm 4.1, Line 1), and if it observes a

free resource, it will attempt to acquire it (Algorithm 4.1, Line 5). If unsuccessful (note that another active entity may attempt to acquire the same resource at the same time), then it picks a random adjacent area and moves to that area via the connecting reachability edge (Algorithm 4.1, Line 8-10).

Algorithm 4.1 Behavior 0

```
1: L1: rs = observe();
2:      if (rs ≠ empty)
3:          for each r ∈ rs
4:              if (r.state == free)
5:                  val = acquire(r);
6:                  if (val == false)
7:                      continue;
8:      select an unvisited neighbor n from graph G_P;
9:      move(n);
10:     go to L1;
```

Behavior B_1**:** This behavior represents the other extreme wherein U sends a request message to *CyS* and waits for a response (Algorithm 4.2, Line 1-2); then it follows the path received in the message (Algorithm 4.2, Line 3-5). In this case, it will always successfully acquire a resource (Algorithm 4.2, Line 10) in the target area as the access is regulated solely by *CyS*.

Algorithm 4.2 Behavior 1

```
1: L1: send_request();
2:      path = receive_path();
3:      while (NOT end of path)
4:          A = next_hop in path;
5:          move(A);
6:      rs ← observe();
7:      if (rs = empty)
8:          for each r ∈ rs
9:              if (r.state == free)
10:                 val ← acquire(r);
```

Behavior B_2**:** In this behavior, U sends a request message to *CyS* and waits for a response (Algorithm 4.3, Line 1-2). Subsequently, it follows the path received in the message. How-

ever, as it moves, it also observes each intermediate A for a free resource (Algorithm 4.3, Line 7); if available, it will attempt to acquire it (Algorithm 4.3, Line 11).

Algorithm 4.3 Behavior 2

```
 1: L1: send_request();
 2:      path = receive_path();
 3:      val = false;
 4:      while (not end of path)
 5:          A = next_hop in path;
 6:          move(A);
 7:          rs = observe();
 8:          if (rs ≠ empty)
 9:              for each r ∈ rs
10:                  if (r.state == free)
11:                      val = acquire(r);
12:                      if (val == true) exit();
13:      if (val == false)
14:          go to L1;
```

Behavior B_3: In B_2, after delivering a path to a resource Ri to U, CyS may find that another resource Rj has been released subsequently which may be closer to U. B_3 is a variation of B_2 wherein U can dynamically accept updated paths from CyS while it is moving (Algorithm 4.4, Line 16), and follows these shorter paths.

We have identified some possible behaviors of active entities above. Clearly, variations of these behaviors (including more complex ones which, for instance, involve active entities cooperating to avoid conflicts) can be defined in our proposed model. We have identified and studied one such cooperative behavior and has shown that it can significantly reduce the time to acquire a resource.

4.3.2 Cyber algorithms

To accommodate the different behaviors (B_1, B_2, and B_3), we have developed both centralized and distributed cyber algorithms. The algorithms assume that each $C_i \in CE$ runs exactly one process denoted by P_i, each physical area A_i is sensed by exactly one cyber

Algorithm 4.4 Behavior 3

```
1: L1: send_request();
2:      path = receive_path();
3:      val = false;
4:      while (not end of path)
5:          A = next_hop in path;
6:          move(A);
7:          rs = observe();
8:          if (rs ≠ empty)
9:              for each r ∈ rs
10:                 if (r.state == free)
11:                     val = acquire(r);
12:                     if (val == true)
13:                         exit();
14:         /*receive updated path from cyber-subsystem.
15:         It is non-blocking receive*/
16:         new_path = receive_path();
17:         if(new_path ≠ empty)
18:             path = new_path;
19:     if (val == false)
20:         go to L1;
```

entity C_i and sensing ranges of cyber entities do not overlap. This eliminates possibilities of two cyber entities sensing the same resource and a physical resource not being sensed by any cyber entity.

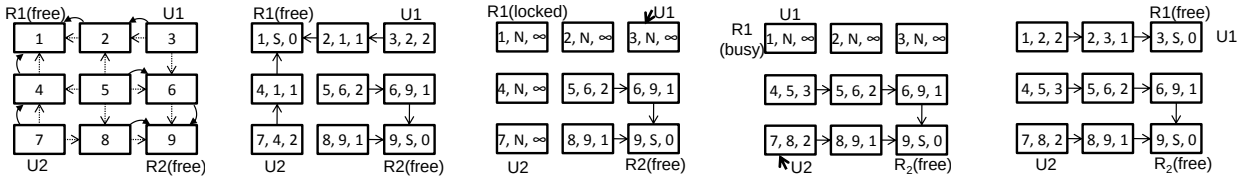
Centralized algorithm

For the centralized algorithm, we assume that there is a central server, and that all sites have routing information to forward packets to and from the server. In this algorithm, the server attempts to maintain up-to-date view of the entire system. For this purpose, it maintains a data structure called *CPSView* which comprises of G_P, G_C, and the states of all areas. This information is updated by messages from processes as the state of the system changes. For example, in Figure 1.1(b), if $U1$ moves from A_4 to A_1, a *lose* message is sent by P_4 indicating that it no longer senses $U1$ and a *sense* message is sent by P_1 when it first

senses $U1$. These messages are forwarded to the server, which updates its local copy of the state variables accordingly.

When an active entity, say U, in area A_i makes a request, the request is forwarded to the server. On receiving this request, the server conducts a breadth first search with A_i as the root in its local copy of G_P to locate the nearest free resource. Note that while the G_C is used for routing, the search is conducted on the G_P. For example, in Figure 1.1(b), if $U2$ makes a request, then the server will initiate a breadth first search by taking A_1 as the root in its stored copy of the graph. In this case, it will find R in A_2 as the nearest resource. When the server finds a free resource, say in area A_j, it sets the local state of the resource as *locked*, computes the path from A_i to A_j in G_P and sends the path back to P_i. On receiving the path from P_i, U can start moving towards A_j. If U acquires the resource on reaching A_j, P_j will send an *acquire* message to the server which changes the state from *locked* to *busy* for that resource.

By setting the state to *locked*, the server excludes the locked resource from future breadth first searches until it becomes free. However, it might be the case that $U2$ may fail to acquire R – this might be due to the fact that $U2$ may itself see another free resource which is closer or some other active entity may have acquired R. To address this, the server uses a *timeout* mechanism. When a path is sent, the server starts a timer. If the server finds that the state of the resource is still *locked* when the timer expires, it changes the state to *free* so that the resource is included in future searches.



(a) *KRL* initial DAG. (b) *SPRA* initial setup. (c) $R1$ is locked. (d) P_7 discovers new path. (e) $U1$ releases $R1$.

Figure 4.1: *(a): KRL Algorithm, (b) to (d): SPRA Algorithm - $S = SELF$, $N = NULL$, triple at each node = (P_i, $ptrR_i$, $height_i$).*

Distributed algorithm

In this section, we present a distributed version of the algorithm. The *KRL* algorithm initializes the system so that the directed edges form a DAG. As the algorithm proceeds, the DAG is modified as the token moves. An important aspect is how the existing DAG is re-used for subsequent searches. Figure 4.1(a) shows the initial DAG which consists of two sink nodes P_1 and P_9 with resources $R1$ and $R2$ respectively (this scenario assumes one cyber entity in each physical area). In the *KRL* algorithm, when entity $U1$ requests a resource and is granted $R1$, the token is passed along from P_1 to P_3, and the edges are reversed as the token moves. This edge reversal results in a new sink at P_3 (but with resource marked as busy). This modified DAG is then subsequently used by others for locating resources. For example, if entity $U2$ in A_7 wants a resource and P_7 picks P_4 as its next hop (which is possible in the KRL algorithm), then the request will be forwarded using the existing edges to P_3. In the context of a *CPS*, the following scenarios in the KRL algorithm must be addressed. First, the location of the token becomes delinked from the location of the resource. In the scenario above, the resource is still in A_1 even though the token has moved to P_3. Furthermore, the user may acquire the resource at A_1 and release it at another location with the token still residing at P_3. Since the time to acquire a resource also includes the time it takes for the user to move to the location of the resource, we must minimize this distance as well. Second, when the token reaches P_3, it is marked as busy. If a user at P_4 requests a resource and this request reaches P_3, it must wait until the token is free even though another free resource ($R2$) exists. This second problem was somewhat alleviated by the variation proposed in [8].

In this dissertation, we have explored two strategies to address the issues discussed above. The first strategy, termed as *KRL_CPS*, is a variation of the *KRL* algorithm wherein we do not perform edge reversal when the token moves. Rather, to keep the root of the tree linked to the location of the resource, we perform edge reversal only when the resource moves. Thus, in the scenario in Figure 4.1(a), the first tree will remain rooted at P_1. Only when

$U1$ moves to A_1 and then moves the resource to another location, edge reversals will occur.

The second strategy, termed *Shortest Path Resource Allocation* (*SPRA*), disregards the existing path information and creates paths on a on-demand basis. *SPRA* maintains a set of trees called $SPTree$s. Each $SPTree_i$ is rooted at P_i where V_i senses a free resource. Each P_i maintains two variables, $ptrR_i$ and $height_i$. $attr_i$ is a tuple $(ptrR_i, height_i)$. Initially, for all P_i, $attr_i = (NULL, \infty)$. Each P_i also maintains a set Nbr_Attr_i which contains the most recent *attr* elements received from the neighboring nodes.

Figure 4.1(b) shows an initial setup showing two $SPTree$s rooted at P_1 and P_9. When a process makes a request, the request is forwarded via the parent pointers to the tree root. For example, when $U1$ located in A_3 makes a request, P_3 will propagate the request to P_1. On receiving this request, P_1 sets $R1.state$ to *locked* and sends confirmation back to P_3 via intermediate child pointers. When $U1$ receives the confirmation, it has to move along the path to reach A_1.

Algorithm 4.5 SPTree management algorithm

```
1:      update attr_i in Nbr_Attr_j;
2:      if (ptrR_j == SELF or ptrR_i == P_j)
3:          /* V_j either senses at least one free resource
4:              or P_i points to P_j itself. */
5:          exit();
6:      else
7:          min_height = min(height_k), attr_k ∈ Nbr_Attr_j;
8:          if (height_j ≤ (min_height + 1))
9:              /*height_j is already minimum.*/
10:             exit();
11:         else
12::=           attr_j0 (P_k, height_k + 1);
13:             broadcast attr_j;
```

$SPTree$s are created and maintained as follows. As soon as P_i senses a free resource, it sets $attr_i$ to $(SELF, 0)$. Whenever $attr_i$ changes, P_i broadcasts the new value to its neighbors. If C_j is neighbor of C_i, on receiving $attr_i$, P_j executes Algorithm 4.5 which first updates Nbr_Attr_j, and then updates $attr_j$ if the value received from P_i provides a

lower cost path to a free resource. When a resource, say $R1$ in A_1 in Figure 4.1(b) is locked, P_1 changes $attr_1$ to $(NULL, \infty)$, and sends this value to its children (which are propagated further). Hence, all the nodes in the tree will set their $attr$ to $(NULL, \infty)$ (shown in Figure 4.1(c)). Subsequently, these nodes connect to trees on a on-demand basis. For example, when P_7 receives a request from $U2$, P_7 will attempt to rediscover a resource. It initiates a breadth first search. The resulting $SPTree$s are shown in Figure 4.1(d). Figure 4.1(e) shows $SPTree$s after $U1$ moves $R1$ to PhA_3 and releases it there.

4.4 Simulation and Results

We use the OMNeT++ Discrete Event Simulation System [40, 41] to simulate the algorithms. It was primarily designed to simulate communication networks, however, because of its flexibility, it has successfully been used to simulate queuing networks, hardware architecture, and other complex IT systems. It provides a component (module) based architecture for models. Components are programmed in C++, and assembled into larger components and models using a high-level language called NED. It also provides extensive GUI support, and due to its modular architecture, the simulation kernel (and models) can be embedded easily into other applications such as Google Maps [42]. It ships with a simulation kernel library, NED topology description language, IDE based on the Eclipse platform, GUI for simulation execution, links into simulation executable (Tkenv), command-line user interface for simulation execution (Cmdenv), utilities (makefile creation tool, etc.), documentation, and sample simulations, etc. Following is a step by step quick overview of how one can use OMNeT++.

- The first step is to define the system model. An OMNeT++ model is build from components which communicate by exchanging messages. Modules can be nested, *i.e.*, several modules can be grouped together to form a compound module. The model has to be mapped to the system into an hierarchy of communicating modules

using the NED language.

- Program the components of the model in C++, using the simulation kernel and class library.
- Provide a suitable omnetpp.ini to hold OMNeT++ configuration and parameters to the model. A config file can describe several simulation runs with different parameters.
- Lastly, build the simulation program and run it. It requires linking the code with the OMNeT++ simulation kernel and one of the user interfaces OMNeT++ provides. There are command line (batch) and interactive, graphical user interfaces.

Once these steps are complete, simulation can be run. Results are written into output vector and output scalar files. These files can be analyzed using the Analysis Tool provided by the Simulation IDE. The result files are text-based, and hence can also be processed with R, Matlab or other tools.

OMNeT++ is flexible enough to be extended with enhanced capabilities such as mobility of nodes. Researchers have developed various extensions to it to simulate specific types of systems. MiXiM [43] is one such extension of OMNeT++ to simulate wireless and mobile networks and provides detailed models of the lower layers of the protocol stack. Our simulation is built on top of MiXiM. Each node in a simulation has two layers, application layer (*appl*) and network layer (*net*). Figure 4.2 shows the architecture of a node in the simulation. The behavior of each entity is coded in *appl*. For example, the *appl* at each cyber entity implements the cyber algorithm being simulated. Each node has a mobility component (*mob*) which is connected to its *appl*. For example, in behavior B_1 (Algorithm 4.2), the $move(A)$ action is implemented by the *appl* sending a *move* message to its *mob* component. When an active entity $U1$ moves a resource $R1$, *appl* of both $U1$ and $R1$ send *move* messages to their respective *mob* components at the same time. We have also simulated the sensing activities by having the *appl* layers of the resources and active entities send periodic

messages announcing their presence. This interaction and movement of active entities and resources can be visualized on the screen during run time.

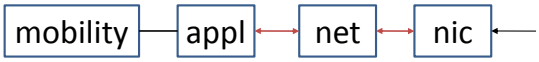


Figure 4.2: *Architecture of an entity.*

In the following discussion, active entity and resource will be referred to as person and wheelchair respectively. We use AT to represent the *Acquire Time*, which is the time elapsed from when a request is made and a wheelchair is acquired, NM represents the total number of messages generated in the network per request, and NH represents the number of physical areas a person needs to move to get a wheelchair. AT, NM and NH are averaged for 100 requests per person. For the experiments, we fixed the time it takes for a person to move from one area to an adjacent one to 3 seconds, and assumed that each person uses a wheelchair for a random amount of time between 20 and 30 seconds. Furthermore, we assume that cyber entities sense the status of the physical area it is located in every 100ms, and the default O_R is 1. We use the 5-tuple $< M, K, B_i, N_P, N_W >$ to represent a system configuration having N_P persons and N_W wheelchairs located in a grid of size $M * K$ (or G_{M*K}) of physical areas and all N_P persons following behavior B_i. $V_{x,y}$ represent the cyber entity located in row x and column y of the grid.

4.4.1 Comparison of KRL_CPS and SPRA

We started by simulating *KRL_CPS* algorithm discussed in Section 4.3.2. In the first scenario of *KRL_CPS*, which we call *KRL-S*, a wheelchair is released at the same location where it was acquired. *KRL-D* refers to a scenario in which wheelchair is released at a random location (which is more realistic). As shown in Figure 4.3, for configuration $< 8, 8, B_1, 6, 3 >$, NH is 18.3 in *KRL-D*, and 12.4 in *KRL-S*. The difference is due to the fact that when the wheelchair moves from its original location, the edges are reversed (hence, a linear chain of parent pointers will be created from its original location to the new location). For example,

in Figure 4.1(b), when $U1$ moves $R1$ to A_3, a tree rooted at P_3 is created. This adds 2 additional hops from A_1 to A_3 in *KRL-D* as compared to *KRL-S* (in which resources are released at the same location). For instance, now when $U2$ makes a request, it has to travel 4 hops to get to $R1$. Also note that in Figure 4.1(b), a free wheelchair is available 2 hops away in A_9. Our *SPRA* algorithm is able to identify such nearby free wheelchairs. The corresponding NH for *SPRA* is 8.3 with wheelchairs released at random locations. As can be seen in Figure 4.3, as N_W is increased (with N_P kept constant), the difference between the performances of the three algorithms reduce. This is due to the fact that with more wheelchairs (*e.g.*, 10 wheelchairs in a G_{8*8}), the trees have smaller depths. As NH is higher for *KRL_CPS* as compared to *SPRA* algorithm, one would expect AT also to be higher. Figure 4.4 shows the performance of these algorithms with respect to AT. As can be seen, *SPRA* outperforms both *KRL-D* and *KRL-S*. We also simulated similar configurations with G_{12*12} and the results follow a similar pattern. However, the performance gain for *SPRA* comes at the expense of increased number of messages. To re-create paths on demand, we have to conduct a breadth first search when a wheelchair is requested. Whereas NM is 31 for *KRL-S* and 47 for *KRL-D*, it is 93 for *SPRA* for configuration $< 8, 8, B_1, 6, 3 >$. However, as N_W is increased (keeping N_P fixed), we find that the cost of re-creating paths drops as it is more likely that nearby resources can be found (and hence, the breadth first search terminates in relatively fewer number of hops). Our simulation show that NM drops from 93 to 61 as N_W is increased from 3 to 6.

4.4.2 Comparison of different behaviors

Next, we wanted to analyze the impact of different behaviors of active entities on AT and NH. In what follows, *SPRA-N* denotes *SPRA* algorithm for behavior B_N where $1 \leq N \leq 3$. We first studied the impact of releasing wheelchairs in random areas on AT and NH by keeping N_W constant and varying N_P. The results are shown in Figure 4.5 and Figure 4.6. As discussed earlier, in B_0 (referred to as *NoCS* in Figure 4.6(a)), a person attempts to visit

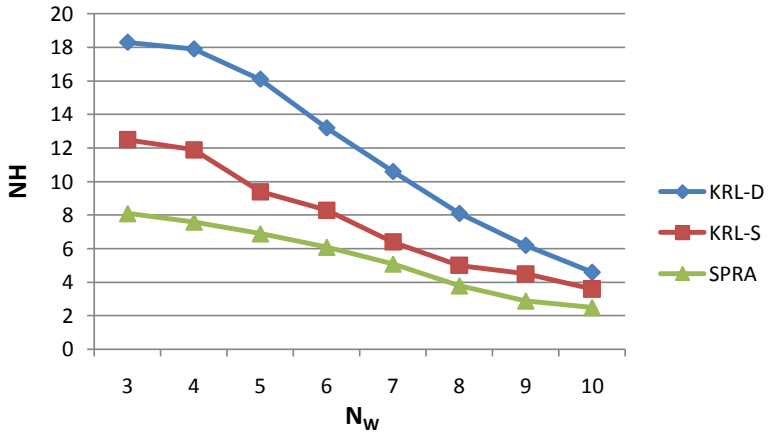


Figure 4.3: *NH vs* N_W *for* $< 8, 8, B_1, 6, N_W >$, $3 \leq N_W \leq 10$.

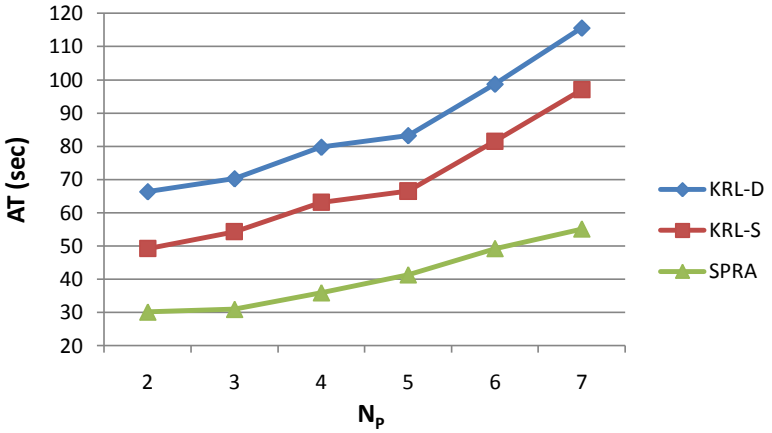
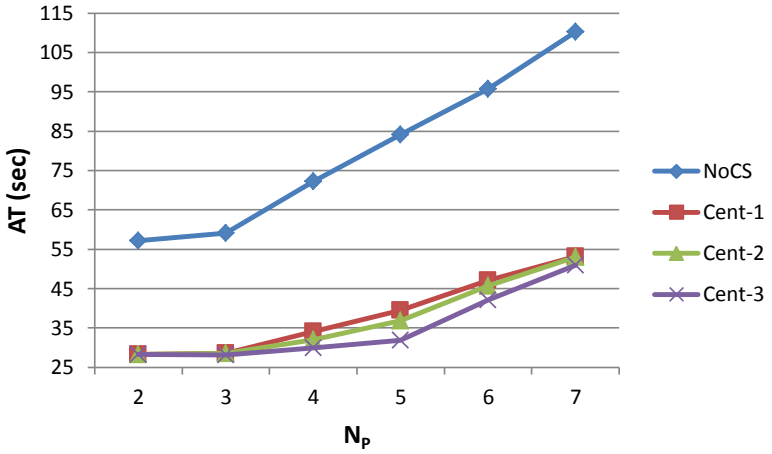


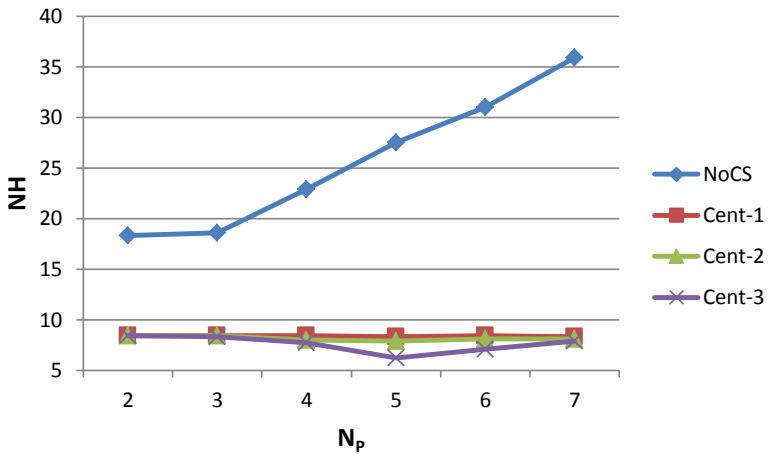
Figure 4.4: *AT vs* N_P *for* $< 8, 8, B_1, N_P, 3 >$, $3 \leq N_P \leq 7$.

areas on its own (without help of *CyS*). This results in a high value of AT. For the other three behaviors, we observed the following: When N_P is 7 and N_W is 3, there is increased competition for wheelchairs. As a result, it is less likely that a person will locate another free wheelchair when it is moving to the location of the free wheelchair initially identified by *CyS* (which it tries to do in B_2 and B_3). Similarly, it is less likely that *CyS* will be able to provide an updated path. Hence, the performance of the three behaviors coincide for this scenario. As the number of persons is decreased (from 7 to 5), there is less competition and the scenarios wherein free wheelchairs can be located by the person or *CyS* become more probable, and *SPRA-3* outperforms *SPRA-2*, which in turn outperforms *SPRA-1*. As N_P is further decreased (say to 2), we find that a free resource will always be available and hence, the initial location identified by *CyS* is most likely to be the nearest one. Hence, the performances again converge. The impact on NH is similar (see Figure 4.6(b)). We observed similar pattern of variation in AT and NH for configurations simulated on G_{12*12}.

To further study the impact of various behaviors, we conducted a set of experiments with localized release of wheelchairs. Figure 4.7 shows the setup of G_{3*17} with four distinct parts of the grid labeled A_1, A_2, A_3, and A_4. Initially, A_1, A_2, A_3 and A_4 has 5, 0, 3 and 3 wheelchairs respectively. We assume that A_2 is similar to an entrance area and all requests are made by persons in this area (we assume a total of 15 persons). We assume that a wheelchair acquired in grid area A_i is released within that grid itself (localized release). Figure 4.8 shows the impact on AT when the distance between areas A_1 and A_2 (dist(A_1, A_2)) is increased from 0 to 6. The scenario shown in Figure 4.7 corresponds to dist$(A_1, A_2) = 0$, and we incrementally move A_2 closer to A_3 in each experiment. In Figure 4.8, we see that AT for B_3 is lower than B_2 and B_1 when A_2 is close to either A_1 or to A_3. This can be explained as follows: Consider the scenario where dist$(A_1, A_2) = 0$, and a wheelchair is released in A_1. Just prior to this moment, assume that a user U in A_2 had requested a wheelchair and was supplied a path to a free wheelchair in A_3 or A_4. In this case, it is likely that *CyS* will find the newly released wheelchair in A_1 closer, and will deliver a shorter path to U. However,

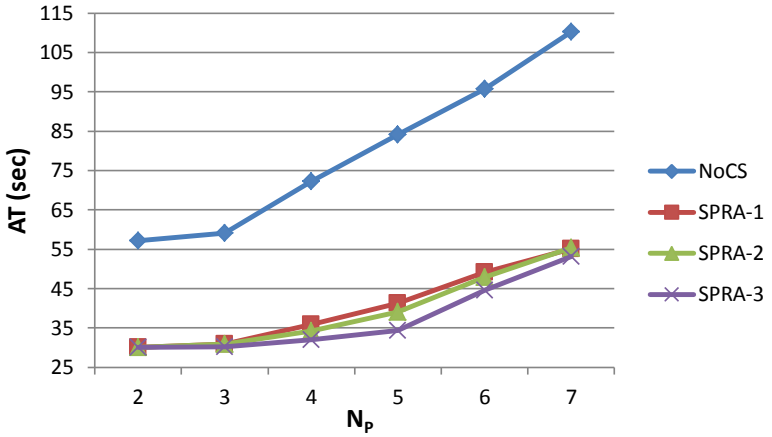


(a) AT vs N_P.

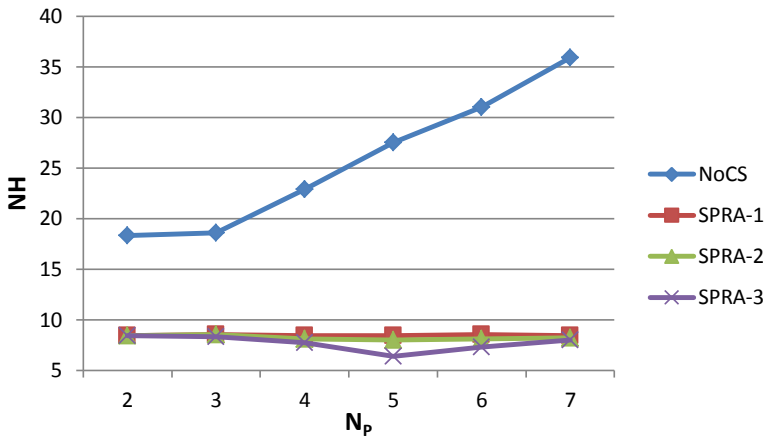


(b) NH vs N_P.

Figure 4.5: *Impact of varying N_P on AT and NH when $N_W = 3$ for G_{8*8} in centralized algorithm.*



(a) AT vs N_P.



(b) NH vs N_P.

Figure 4.6: *Impact of varying N_P on AT and NH when $N_W = 3$ for G_{8*8} in distributed algorithm.*

as dist(A_1, A_2) increases, this scenario becomes less likely and hence the performance of B_3 converges to that of B_2 (see Figure 4.8). However, as dist(A_2, A_3) decreases, the scenarios with free resources in A_3 or A_4 become likely, and again B_3 starts performing better than B_2. The results for the centralized algorithm follow a similar pattern.

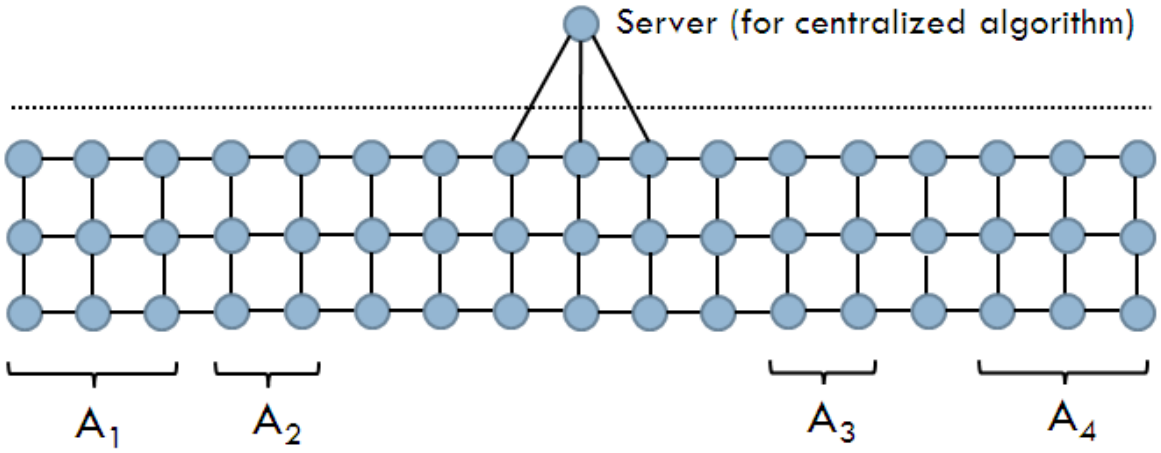


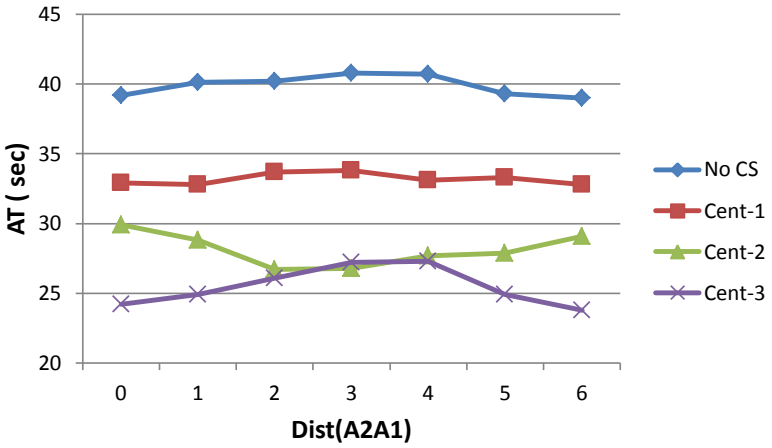
Figure 4.7: *Setup of* G_{3*17}.

4.4.3 Impact of Server location

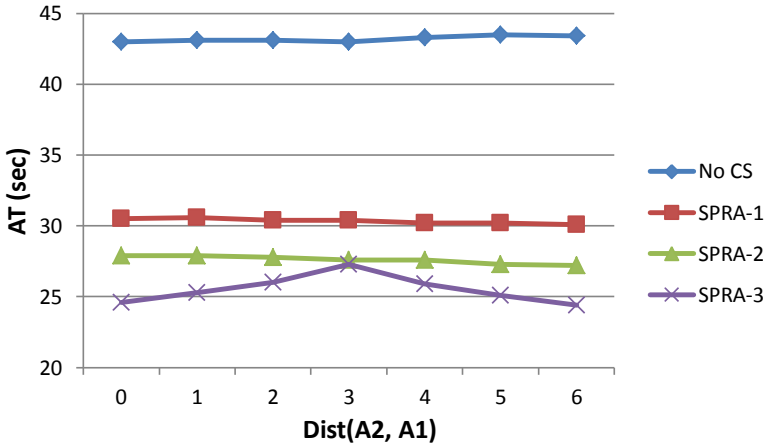
In this setup, we changed the server location for the centralized algorithm in Figure 4.7 such that each time server communicates only with $V_{1,n}$, $1 \leq n \leq 17$. We noticed that as the server distance from grid A_2 (from where persons make requests) increases, AT also increases. This is due to the fact that it takes longer for each request to travel to the central server. This shows that server location should be chosen carefully if one decides to opt for a centralized solution. Figure 4.9 shows the detailed results.

4.4.4 Impact of O_R

In this setup, we increased O_R of each person from 1 to 8 for configurations $< 8, 8, B_i, 5, 3 >$, $1 \leq i \leq 3$. The results for *SPRA* are shown in Figure 4.10. The performance of B_1 is not impacted by O_R. The performance of B_2 improves as O_R is increased from 1 to 4 – this



(a) *AT* vs dist(A2, A1) for centralized algorithm.



(b) *AT* vs dist(A2, A1) for SPRA algorithm.

Figure 4.8: *Impact of localized released of wheelchairs on AT.*

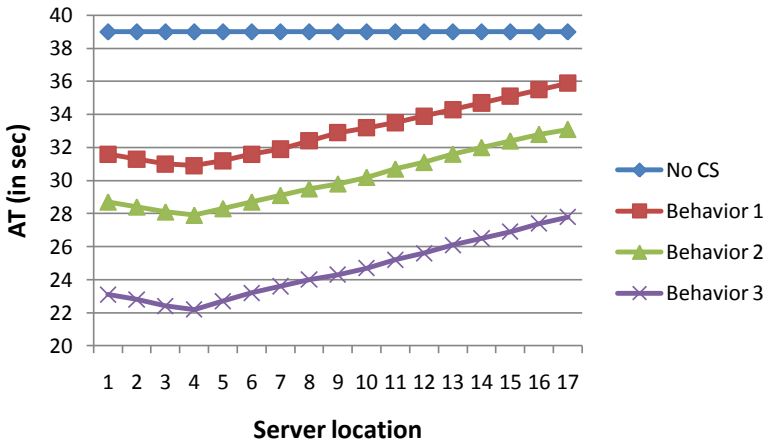
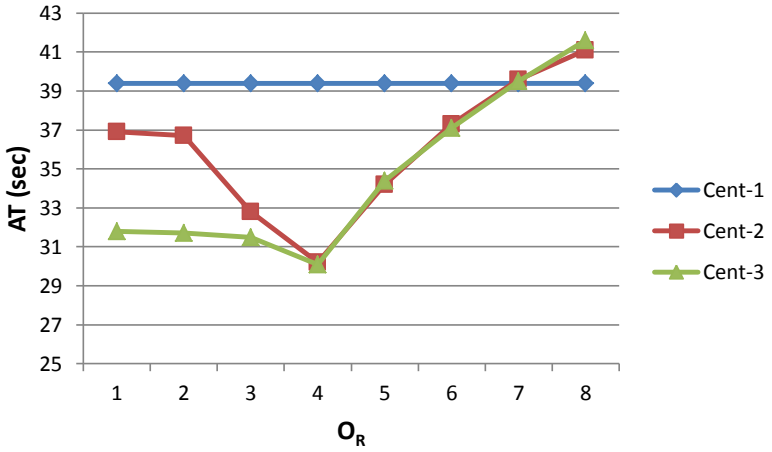


Figure 4.9: *AT vs server location.*

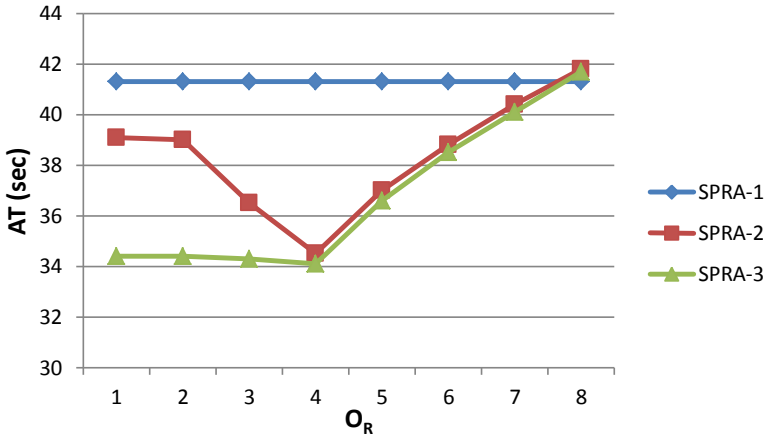
is due to the fact that a person can observe more areas and hence the chances of finding a nearby free wheelchairs increase. However, as O_R is increased further, performances of B_2 starts degrading because the observation zones of the users overlap a lot. Hence, there are more chances that whenever a wheelchair becomes free, multiple users might observe it and deviate from their original paths towards this free wheelchair. Since only one of them will be successful, others will have to incur additional hops. The performance of B_3 show a similar pattern except that when O_R is increased from 1 to 4, we do not see much change. In this case, we find that *CyS* is able to provide quick updates of newly freed wheelchairs which are close.

4.4.5 Impact of cyber-subsystem delay (S_D)

In the experiments above, we assumed that the cyber subsystem delay, which is the time gap between two sensing activities, as 100ms. Note that the value of state variables ($rs.state$, $A.ae$ and $A.rs$) available to processes correspond to the last sensing activity. Hence, as S_d increases, the probability that the state information available to *CyS* is stale increases. We increased S_D from 100ms to 3500ms and analyzed the impact on the performance of various behaviors. Since the allocation is completely controlled by *CyS* in B_1, the performance

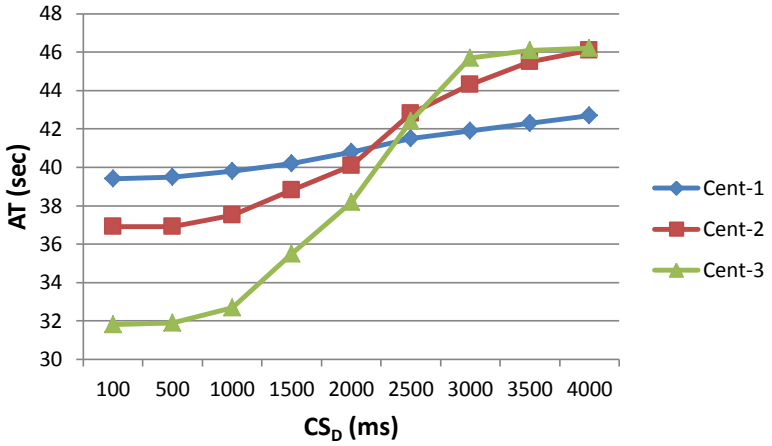


(a) AT vs O_R for centralized algorithm.

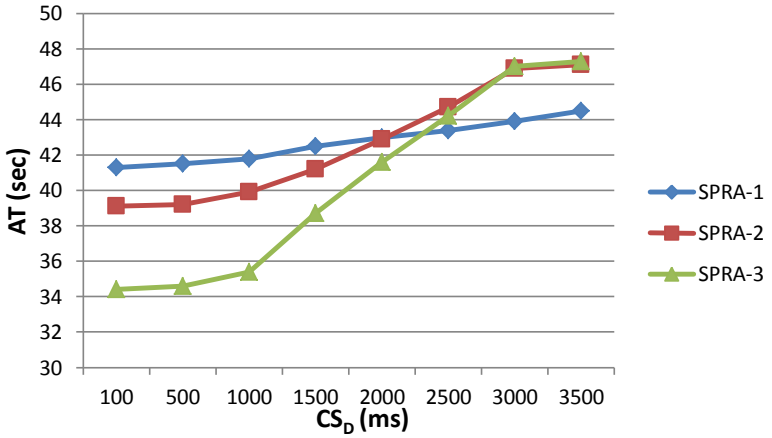


(b) AT vs O_R for SPRA algorithm.

Figure 4.10: *Impact of increasing O_R on AT.*



(a) AT vs cyber-subsystem delay for centralized algorithm.



(b) AT vs CS_D for SPRA algorithm.

Figure 4.11: *Impact of increasing CS_D on AT.*

degrades linearly as S_D is increased. However, in B_2 and B_3, it is possible that a person, say $U1$, may acquire a wheelchair $R1$ in area A_1 which was not allocated to $U1$ by CyS. However, this fact will only become known to CyS during the next sensing activity. Increasing S_D will increase this period, and may result in the cyber algorithm making decisions based on stale information. As a result, we find that the performance of B_2 and B_3 degrades significantly as S_D is increased. Figure 4.11 shows the results for configurations $< 8, 8, B_i, 5, 3 >$, $1 \leq i \leq 3$.

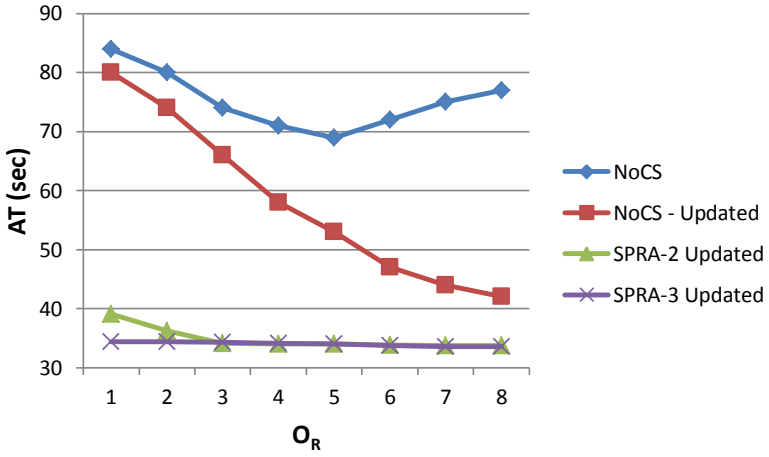


Figure 4.12: *Impact of persons cooperating each other on AT for* G_{8*8}, $N_P = 5$ *and* $N_W = 3$.

4.4.6 Cooperative user behavior

We also experimented with a scenario in which the users cooperate with each other. When a user $U1$ is using a wheelchair $R1$ and another user $U2$ observes $U1$ using $R1$, $U2$ assumes that $U1$ will release $R1$ at some point of time. In our existing algorithm, $U2$ will wait for a notification of the release and then move towards the location where $R1$ has been released. From our experiments, we found that the time spent in moving to the resource after the release notification adds significantly to the acquire time (AT). We therefore modified the behavior as follows: If $U2$ finds that $U1$ is using $R1$, $U2$ starts following $U1$ and as soon as $U1$ releases $R1$, $U2$ will attempt to acquire it. This overlaps the time when $R1$ is being

used with the time spent in moving after the resource is released. Furthermore, if a third person $U3$ observes that $U2$ is moving towards $U1$, $U3$ assumes that $U2$ wants to acquire $R1$ after $U1$ will releases it. In this case, $U3$ does not follow $U1$. In our earlier experiment (Figure 4.10), AT increased when observation radius is increased beyond a threshold due to increased competition. Figure 4.12 shows the results for $NoCS$ as well as the *updated SPRA-1*, *SPRA-2*, and $NoCS$ behaviors in which persons cooperate as described above. As one can see, the cooperative behavior is able to resolve the conflicts due to competition and AT decreases as radius is increased.

4.5 Summary

In this chapter, based flat spatial model, we presented algorithms for the mutual exclusion problem in CPS. Each algorithm had two components, one describing the behavior of users in *PhyS* the other describing the cyber algorithm. We identified several characteristic of a CPS which make solutions for TDS inapplicable to a CPS. We simulated all the presented algorithms using OMNeT++. The results provide suggestions on the best algorithm to use in different scenarios. For example, the results show that when fewer resources are present, it might be best to rely completely on *CyS*; otherwise, participation of users in locating resource can improve performance.

Chapter 5

Predicate Detection and Model Checking

The problem of predicate detection has traditionally been studied in the context of TDSs, in which predicates are defined over variables of processes constituting underlying TDS. Predicates in the context of CPSs are different in the sense that predicates are defined over actions performed by users in *PhyS*, and are influenced by time and location. In this chapter, we

- introduce the concept of predicates in CPSs
- present model checking techniques for CPSs using UPPAAL [44–46].
- present a centralized and a distributed algorithms which generate alerts when a predicate is violated, and also provides useful information to the users to minimize instances of predicate violation.

5.1 Introduction

In concurrent systems, *await* statements of the form $\langle await\ (B)\ S; \rangle$ are used to ensure that condition B is *true* before a sequence of statements S is executed atomically [47]. In a CPS, it is analogous to a situation when the user behavior is disciplined. A user will perform an action only after getting permission from *CyS*. If *CyS* finds that the action may cause any constraint violation, it will instruct the user not to perform the action. However, this is not possible in practice. Users in *PhyS* perform actions asynchronously with respect to *CyS*, *i.e.*, users perform S and it is the responsibility of *CyS* to ensure B. The predicate detection algorithm running on *CyS* must reactively generate alerts when B becomes $false$ as a result of performing S by any user; and proactively control the behavior of the users by alerting them from doing actions which may cause B to become $false$.

In other words, the problem of predicate detection in CPS can be though of enforcing constrains on actions in *PhyS*, and detecting when they are violated. For example, consider the task of transferring a patient from one location to another. A patient may be required to be accompanied by a medical staff with specific medical equipment during the transfer. *CyS* can provide the capability of enforcing such constraints by automatically detecting the presence of appropriate medical staff and equipment in the vicinity of the patient at all times during the movement, and issuing alerts when these conditions are violated. Similar constraints regulating presence of medical personnel and equipment in various units can be enforced via *CyS*.

There are many other examples of physical system which require enforcement of similar constraints and conditions. [31] discusses a context aware workflow system in a smart factory and constraints to be enforced, such as *the usage time of a tool should not exceed a given time*. [48] discusses various constraints for a smart home such as *if a person stays in the living room for more than 2 seconds and the living room's light is turned off, then the lights should be turned on in 5 seconds*. Similar constraints are discussed for a smart construction site in [35] and for an intelligent water distribution network in [49].

The key idea in such applications, as discussed above, is to enforce constraints and detect their violation in *PhyS*. This problem has been abstracted as a predicate detection problem in a traditional distributed system (TDS) [50, 51]. Although one can extend such algorithms for a TDS to a CPS, they appear on one end of the spectrum where all actions are controlled entirely by *CyS*. In a CPS, however, we may want to allow the possibility of entities in both *CyS* and *PhyS* to cooperate/interact in solving a problem. Allowing this requires one to address several challenges, some of which are discussed below:

In a TDS, the predicates are defined in terms of variables of different processes of the system. For example, if CS_i is a predicate which is *true* when the process i is in critical section, then $CS_1 \wedge CS_2$ represents a predicate that process 1 and process 2 are in the critical section at the same time. The values of the variables in such predicates are completely controlled by events such as message receive event, or events internal to a process.

In a CPS, *CyS* may keep track of the state of *PhyS* via a set of variables which are used to define predicates. For example, we may use $Loc.Nurse$ to represent the set of nurses present in a location *Loc*. Then, in Figure 1.2, $NurseStation1$ contains two locations A_2 and A_3 monitored separately, and $NurseStation1.Nurse = \{N_2, N_6\}$), and the predicate $|NurseStation1.Nurse| \geq 1$ represents a constraint which states that there should be at least one nurse at $NurseStation1$ at all times. Due to sensing delays (sampling frequency of sensors) or lost sensor values, there may be a difference between the actual state of *PhyS* and the corresponding variables maintained in *CyS*. For instance, assume that the nurse N_6 moves from A_3 to A_2 at time t. Due to sensing delay S_d, $A_2.Nurse$ may not reflect the presence of N_6 until a later time $t + S_d$. Such delays may result in inconsistencies when applying traditional algorithms directly to a CPS. For example, a traditional global state recording algorithm used for a CPS would form a global state by collecting states of individual areas. In a CPS, sensing delays and unsynchronized clocks may lead to a situation where both A_2 and A_3 report presence of the same nurse. Similarly, we must contend with an opposite scenario where both stations report absence of N_6. This may be

caused due to unsynchronized sampling of states or due to the fact that N_6 may be in an intermediate area which is out of the sensing ranges of both A_2 and A_3.

Predicates in a CPSmay have additional consistency constraints based on location. Locations may have an hierarchy associated with them (e.g., $ICU1$ is contained in $Floor1$, and $Floor1$ is contained in $Hospital1$), and one must ensure consistency of their corresponding state variables. For example, in Figure 1.2, since N_1 is *located* in $ICU1$, it counts towards $ICU1.Nurse$; in addition, it must also be included in $Floor1.Nurse$ and $Hospital1.Nurse$. Predicates may also have a notion of time and location associated with them. Examples of such predicates include *C1: if there is more than one patient in ICU, than there must be at least one nurse in ICU*, and *C2: if a patient presses an emergency button, then a nurse should attend the patient within one minute.*

The issues discussed above may significantly affect the way predicates are expressed and evaluated in CPSs. Several other traditional distributed algorithms such as mutual exclusion [25], constructing global snapshots [35], event ordering [36, 37], and termination detection [38, 39], have been explored in the context of pervasive systems and CPSs. Time and location aware predicate models and their detection algorithms have also been proposed [11, 52, 53], however, they have their own shortcomings as discussed in Section 5.2.

5.2 Related Work

In a TDS, the predicates can broadly be classified into two categories: stable predicates and unstable predicates. Once evaluated to *true*, a stable predicate never turns $false$. Deadlock and termination are examples of stable predicates in a TDS and can be detected using global state recording algorithms such as presented in [54–56]. The central idea in these algorithms is that they record a consistent global snapshot of the system and evaluates the predicate over that snapshot. If the the predicate is evaluated to *true*, then it can be inferred that the predicate is *true* at the end of the algorithm; and if the predicate is evaluated to $false$

at the end of the algorithm, then it can be inferred that it was also $false$ at the beginning of the algorithm. This does not work for evaluating an unstable predicate because it may be *true* only between two snapshots, and not when the snapshot is recorded. The critical section predicate discussed in Section 5.1 is an example of unstable predicate. Garg and Waldecker discuss various techniques to detect unstable predicates in [50, 51].

Predicates in CPS are unstable in nature because physical entities keep moving inside *PhyS* causing the values of variables maintained in *CyS* to change and hence the values of the predicates. Unstable predicate model and their detection algorithms for TDS as presented in [50, 51] are not suitable for CPS given the challenges discussed in Section 5.1. In recent years, due to growth in the field of location based services, various models have been proposed to express location and time based predicates [11, 52, 53]. Chandran and Joshi's predicate model [11] supports a novel spatial model which we have discussed in Section 3.2.2. Their predicate model captures the notion of time as well; however it has several drawbacks. First, their spatial model lacks the concept of reachability edges which is required as will be discussed in Section 3.2.2. Second, the types of predicates they discuss are more related to role based access control ($RBAC$); for example, whether a particular role is enabled at a particular location at time t.

Ardagna et al. [52] also proposes a location based predicate model for $RBAC$. Their spatial model is not as expressive as Chandran and Joshi's spatial model. Moreover, their model has very limited expressive power and can express only following conditions:

- whether user is located within/outside an area.
- whether the distance between user and an entity is within certain interval.
- whether user's speed falls within certain range.
- whether the number of users currently in area falls within a certain range.

For pervasive systems, [53] proposes three predicate detection algorithms having varying degrees of accuracy. Some of the predicates they detect are similar to the predicates we

present such as *number of persons in room1 = N*. However, their algorithm does not proactively alert users from doing actions which may violate a predicate, and they do not define a concrete spatial model of *PhyS*.

Yao et al. [57] presents an event model which can express events with logical and temporal operators. It also provides capability to express movement of objects. Given events E_1, E_2, time t_1, t_2, and an integer n, the following operators can be used to express predicates:

- $E_1 \wedge E_2$: Conjunction of two events E_1 and E_2 without occurrence order.
- $E_1 \vee E_2$: Disjunction of two events E_1 and E_2 without occurrence order.
- $\neg E_1$: Negation of E1
- $(E_1; E_2)$: E_1 occurs followed by E_2.
- $window(E_1, t_1)$: Event E_1 occurs for time period t_1.
- $window(E_1, n)$: Event E_1 occurs n times.
- $within(E_1, t_1)$: Event E_1 occurs within less than t_1
- $within(E_1, t_1, t_2)$: Event E_1 occurs within interval t_1 and t_2.
- (E_1, t_1): Event E_1 occurs at time t_1 [system time].
- E_1^* Every occurrence of E_1.
- $during(E_1, E_2)$: Event E_2 occurs during event E_1.

Using above operators, one can specify various other predicates such as whether an object is entering or leaving an area, whether an objects is entering or leaving proximity of another object, whether a person is acquiring or releasing an object, and whether an object is touching/next to another object. This model provides a rich set of operators, however its spatial model is not very expressive. In the next section, we define time and location aware predicates for the CPSs, which overcomes the shortcomings of the existing models.

5.3 Predicates in CPS

Based on the CPS model which we have defined in Chapter 3, we define two types of predicates in CPS: Location predicates and Service predicates.

Location predicates impose constraint on the number of physical entities present in a physical area. An example of such a predicate is *there should always be a nurse present at the nurse station*. Another example, which involves a condition is *there should always be a nurse in the ICU if there is at least one patient in the ICU*. As we have discussed, predicates in CPS need to be time and location aware.

Service predicates involves the time at which an action is initiated by an active entity and the time at which its consequences need to occur. For example, a patient requests for a doctor at time t, and a doctor should be available to the patient within 5 minutes, *i.e.*, within time interval t and $t + 5$. For this purpose, we classify active entities into two categories: *providers* and *requesters*. Provider entities provide services to requester entities. For example, a nurse is a provider and a patient is a requester in the context of hospital CPS. We illustrate both location and service predicates using examples below. The example also illustrate the way we write the predicates.

Example 5.3.1 *(Location Predicate — $pred_1$): There must be at least one nurse available at the nurse station (NS) in the hospital:*

$\forall ns \in NS, |ns.Nurse| \geq 1$

Example 5.3.2 *(Location Predicate — $pred_2$): There must be at least one nurse available in the ICU if there is at least one patient in the ICU:*

$\forall icu \in ICU, |icu.Patient| \geq 1 \rightarrow |icu.Nurse| \geq 1$

Example 5.3.3 *(Location Predicate — $pred_3$): There must be at least one doctor available at each floor:*

$\forall f \in Floor, |f.Doctor| \geq 1$

Example 5.3.4 *(Service Predicate — $pred_4$): If a patient p needs a nurse (assuming that*

the patient presses a button), a nurse n should be available to the patient within t_n minutes:

$p \in Patient, p.location = A$

t: $p.needsNurse = true$

$\exists n \in Nurse$

$t \leq t' \leq t + m$: $n.location = A$

Example 5.3.5 *(Service Predicate — $pred_5$): If a patient p needs a doctor (assuming that the patient presses a button), a doctor d should be available to the patient within t_d minutes:*

$p \in Patient, p.location = A$

t: $p.needsDoctor = true$

$\exists d \in Doctor$

$t \leq t' \leq t_d$: $d.location = A$

The examples which have discussed imposes stringent temporal requirements, such as, *there **must** always be a nurse in the nurse station*, or *the doctor should be available within n minutes*. If we relax these requirements, we get a completely new set of predicates, which imposes weaker temporal requirements. We relax these requirements either by allowing a predicate to remain false for some amount of time, or by increasing the time within which provider should be available to the requester. For the first case, we write next to the predicate *false for t minutes*, indicating that once the predicate becomes false, it can remain false for at most t minutes, and must become true within t minutes. The higher the time, the weaker the predicate is. CPS predicates are represented in $CPSML$ using following constructs:

⟨predicate⟩→ ⟨location⟩ | ⟨service⟩

⟨location⟩→ ⟨complex⟩ | ⟨complex⟩ '− >' ⟨complex⟩

⟨complex⟩→ ⟨basic⟩ | ⟨basic⟩ (**and**|**or**) ⟨basic⟩

⟨basic⟩→ (**for all** ⟨id⟩ **of type** ⟨id⟩ **.** ⟨id⟩)?

| ⟨id⟩ **.** ⟨id⟩ | ⟨comparision_op⟩ ⟨int⟩

⟨service⟩→ ⟨comparision_op⟩→ '<' | '>' | '<=' | '>=' | '==' | '! ='

⟨id⟩→ **(‘a’..‘z’|‘A’..‘Z’|‘_’) (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’)***

⟨int⟩→ **(‘0’..‘9’)+**

In the next section, we briefly discuss UPPAAL model checker and then show how it can be used to model check a CPS with an example.

5.4 Uppaal Model Checker

UPPAAL [44–46] is a tool to simulate, verify and validate real time system modelled as a network of timed automata [58], which are finite state machines [59] extended with clock variables. UPPAAL provides a modeling language to specify system's model in the form of networked timed automata. It further allows to extend timed automata with discrete variables which are part of the system state. The system state is set of all the locations (current states) of constituent automata, the clock values, and discrete variables' values. An automaton may re an edge (transition) separately which may depend on the values of clock(s) and/or discrete variable. It may also synchronise with another automaton. Firing of an edge usually leads the system to a new state. UPPAAL also provides a query language which is a subset of of TCTL (timed computation tree logic) [60] and used to specify properties to be checked.

We illustrate capabilities of UPPAAL using a Train Gate system which is distributed with UPPAAL. Train gate is a railway control system which controls access to a bridge, which is shared by several trains and each train needs an exclusive access to the bridge. The system is dened as a number of trains (6 in this example) and a gate controller, which provides trains an exclusive access to the bridge. There are timing constraints on the train before entering the bridge because of following two reasons: a) a train can not be stopped instantly, b) restarting a train takes some time. When a train approaches the bridge, it sends a *appr*! signal to the controller. Thereafter, it has 10 time units to receive a stop

signal. This allows it to stop safely before the bridge. After these 10 time units, it takes further 10 time units to reach the bridge if the train is not stopped. If a train is stopped, it resumes its course when the controller sends a *go*! signal to it after a previous train has left the bridge and sent a *leave*! signal.

5.4.1 Train Gate Modeling in Uppaal

In Uppaal, each automaton is defined using a template. Train gate model specifies two templates:

- Train template as shown in Figure 5.1(a).
- Gate controller template as shown in Figure 5.1(b).

The model may define global variables and clocks which are shared among all templates. Each template may in turn define local variables and clocks for its exclusive use. The global declaration for in train gate system is shown in Listing 5.1.

```
const int N = 6;          // # trains
typedef int[0,N-1] id_t;
chan     appr[N], stop[N], leave[N];
urgent chan go[N];
```

Listing 5.1: *Global declarations for train gate model*

Train Template

Train template has a local clock `x`, an identity *id* and has following ve locations:

- `Safe`: It is the initial location of a train indicating that the train is not approaching the bridge yet. A train can be in this location for indefinite amount of time since this location has no invariant. When a train approaches, it synchronises with the controller using channel synchronisation *appr*[*id*]! and the train transitions to `Appr` location.

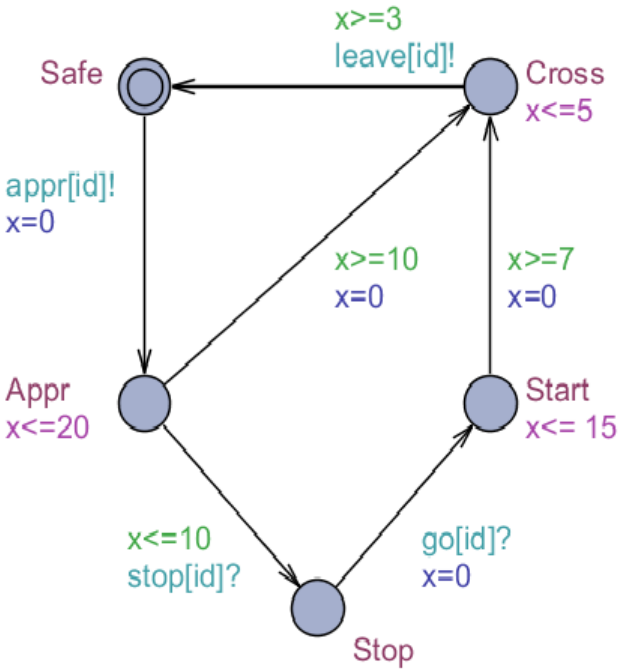
Upon this synchronization, train's clock `x` is reset. The controller has corresponding $appr[id]!$, which enqueues identity of the train.

- `Appr`: The invariant of this location is $x \leq 20$, which means has that the location must be left within 20 time units. There are two outgoing transitions guarded by the constraints $x \leq 10$ and $x \geq 10$:
  - $x \leq 10$: The train can be stopped. In this case, the train transitions to `Stop` location. Transition to `Stop` is synchronised with $stop[id]?$. When the controller decides to stop a train, it decides which one (id) and synchronises with $stop[tail()]!$ ($tail()$ is a queue operation as shown in Listing 5.2.
  - $x \geq 10$: The train can not be stopped. In this case, the train transitions to `Cross` location.

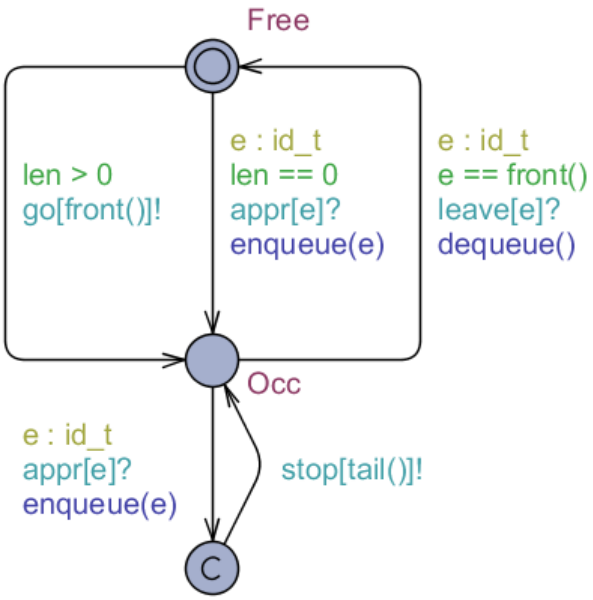
  At exactly 10, both transitions are enabled, which allows us to take into account any race if there is one.
- `Stop`: Similar to `Safe`, the location `Stop` has no invariant implying that a train may be stopped for an indefinite amount of time. At this location, the train waits for the synchronisation $go[front()]?$.
- `Start`: This location has the invariant $x \leq 15$, and one outgoing transition with the constraint $x \geq 7$ implying that a train is restarted and reaches `Cross` location between 7 and 15 time units non-deterministically.
- `Cross`: This location is similar to `Start` location in the sense that the train leaves this location between 3 and 5 time units after entering it.

Gate Template

The gate controller template is shown in Figure 5.1(b). Its local declaration consists of a queue and queue related methods (see Listing 5.2).



(a) Automaton representing a train which needs exclusive access to a shared resource (bridge)



(b) Automaton representing a controller gate which makes sure that only train access the bridge at a given point of time

Figure 5.1: *Train and gate controller automata*

The controller has three locations `Free`, `Occ`, and `C`. The `Free` is the starting location of the controller, implying that the bridge is free. There are two outgoing transitions from this location depending whether the queue is empty. If the queue is empty then the it waits for approaching trains with the $appr[e]?$ synchronisation. When a train is approaching, it is added to the queue using $enqueue(e)$. On the other hand, if the queue is not empty, then the rst train on the queue is restarted with the $go[front()]!$ synchronisation.

In the `Occ` location, the controller waits for the running train to leave the bridge ($leave[e]?$). If other trains are approaching ($appr[e]?$), they are added to the queue ($enqueue(e)$) and stopped ($stop[tail()]!$). When a train leaves the bridge, the controller removes it from the queue using $dequeue()$ method.

```
id_t list[N+1];
int[0,N] len;
// Put an element at the end of the queue
void enqueue(id_t element)
{
        list[len++] = element;
}
// Remove the front element of the queue
void dequeue()
{
        int i = 0;
        len -= 1;
        while (i < len)
        {
                list[i] = list[i + 1];
                i++;
        }
        list[i] = 0;
}
// Returns the front element of the queue
```

```
id_t front()
{
   return list[0];
}
// Returns the last element of the queue
id_t tail()
{
   return list[len - 1];
}
```

Listing 5.2: *Local declaration for gate*

5.4.2 Train Gate Verification

Once a model is specified, it needs to be checked against the requirements. UPPAALprovides a query language to specify the requirements, which is a simplied version of TCTL. UPPAAL allows us to check following three properties of a model:

- Reachability: It checks whether a given state is reachable. It is represented as $E <>$ *property*. For example, $E <> Train(1).Cross$ represents that train 1 can cross the bridge.

- Safety: It is of the form "something bad will never happen" and is represented as $A[]$ *property*. For example $A[]$ $forall(i : id_t)$ $forall(j : id_t)$ $Train(i).Cross$ && $Train(j).Cross$ *imply* $i == j$ implies that there is never more than one train crossing the bridge at any time instance. The deadlock can be checked using $A[]$ *not deadlock*.

- Liveness: It is of the form " something will eventually happen" and is represented as$A <>$ *property*. For example $A <> Train(0).Appr -- > Train(0).Cross$ implies that whenever a train (in this case 0) approaches the bridge, it will eventually cross.

5.5 Model Checking of CPS using Uppaal

Given the following:

- user behavior,
- instances of users and resources, and
- physical infrastructure specification

the objective of model checking a CPS is to find out whether the system satisfy a predicate p? There may be instances in which the system does not satisfy p, and in those instances, we can choose one or more of the following options:

- change number of users,
- change user behavior,
- weaken the predicate, or
- employ a cyber system

In the following, we explain how UPPAAL is used to model physical infrastructure and user behavior.

5.5.1 Modeling Physical Infrastructure

The physical infrastructure is modeled as an UPPAAL template as follows:

- Identify fine grained areas and reachability edges.
- Corresponding to each fine grained area A_i, create a location with name A_i.
- If for two fine grained areas A_i and A_j, R_{ij} exist, then draw an edge from location A_i to A_j and an edge from location A_j to A_i.

- Identify all types of active entity and resource types. Corresponding to each active entity type τ_{ae}, declare a global `int` variable for each fine and coarse grained area which represents total number of instances of type τ_{ae} located in that area. Similar global variables should be created for each resource type.

Following this, we model the behavior of active entities using guards on the edges of the automaton.

5.5.2 Modeling Active Entities' Behavior

Active entities' behavior consists of actions performed by them inside the physical infrastructure. Depending on the time, and certain event, they move from one area to other. This can be viewed as an active entity moving from a location to another in physical infrastructure template. If we write appropriate guards for location transitions, and update the global variables in physical infrastructure template, we obtain a behavior of an active entity. For example, the starting location in Figure 5.2 is `Start`, and the nurse enters $Floor1$ at $t = 7150$. As soon as the nurse reaches location A_{10}, the corresponding variables are updated. Since the behavior of each active entity is different, we create a physical infrastructure template corresponding to each active entity, and depending on individual's behavior, label each edge with appropriate guards. Partial behavior of a nurse is shown in Figure 5.2.

5.5.3 Specifying Set of Predicates

The CPS predicates which we have defined earlier in this chapter require that they must always be *true*. This suggests that they should be represented safety properties in UPPAAL. The five predicates which we discussed in Section 5.3 are specified in UPPAAL's query language as follows:

- $A[]\ NS1.N >= 1$ represents that number of nurses at $NurseStation1$ must always be

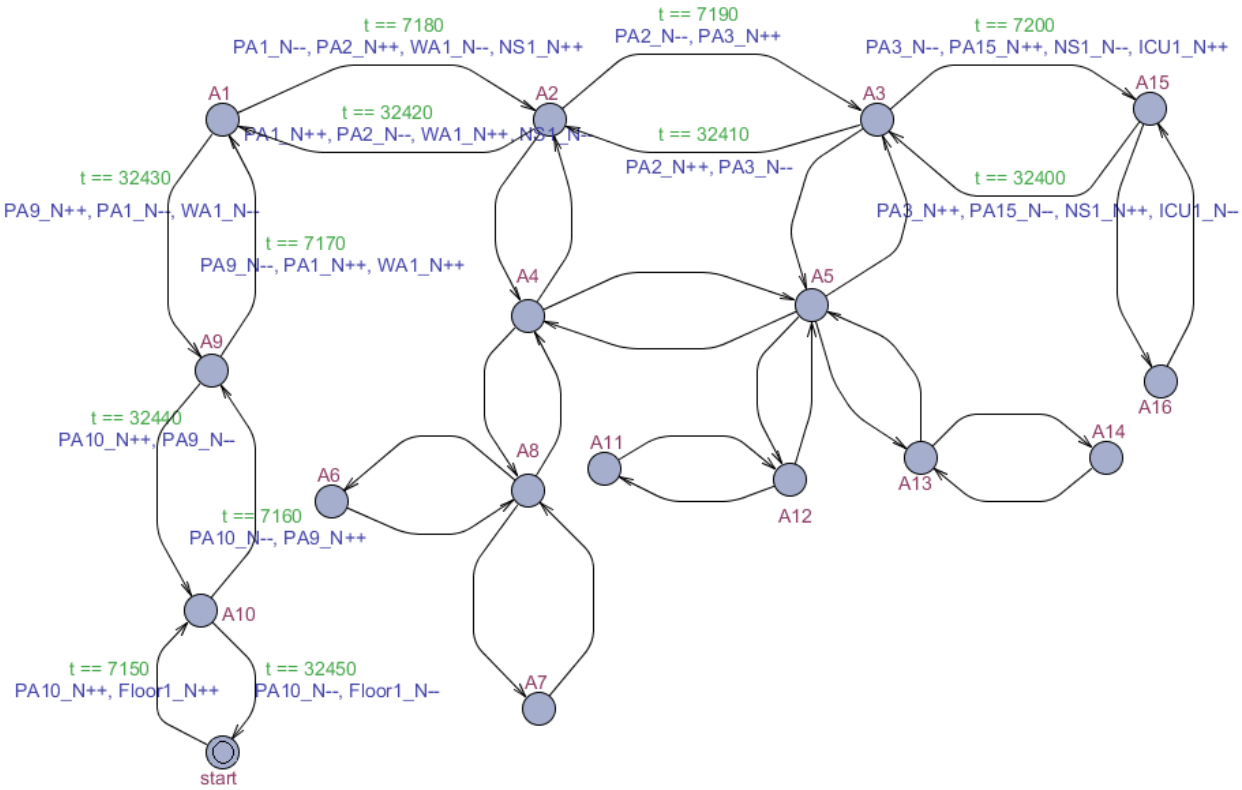


Figure 5.2: *A nurse's behavior modeled in* UPPAAL *for* CPS *shown in Figure 1.2*

greater than or equal to 1.

- $A[]\ ICU1.P >= 1\ imply\ ICU1.N >= 1$ represents that if number of patients in $ICU1$ are greater than or equal to 1, then there must be at least 1 nurse in $ICU1$.

- $A[]\ Floor1.D >= 1$ represents that there must always be at least one doctor at $Floor1$.

- $needNurseIcu1 == true\ imply\ t1n <= 240\ \&\&\ A15.N >= 1$ implies that when a patient located in $ICU1$ (in area A_{15}) needs a nurse's service, then the a nurse should be available to the patient within $t_n = 4$ minutes (or 240 seconds). We model this as setting a clock $t1n$ to 0, and waits for $A15.N$ to be greater than or equal to 1 before the clock hits 240 seconds. We assume that when the nurse's location is the same as the patient's location, then the nurse is servicing the patient.

- $needDoctorIcu1 == true\ imply\ t1d <= 240\ \&\&\ A15.D >= 1$ implies that when a patient located in $ICU1$ (in area $A15$) needs a doctor's service, then the a doctor should be available to the patient within $t_d = 4$ minutes (or 240 seconds). We model this as setting a clock $t1d$ to 0, and waits for $A15.D$ to be greater than or equal to 1 before clock hits 240 seconds. We assume that when the doctor's location is the same as the patient's location, then the doctor is servicing the patient.

In order to verify that all the predicates are satisfied, we create a larger UPPAAL property by combining individual properties using && operator as follows:

$A[]\ NS1.N >= 1\ \&\&\ ICU1.P >= 1\ \&\&\ imply\ ICU1.N >= 1\ Floor1.D >= 1\ \&\&\ needNurseIcu1 == true\ imply\ t1n <= 240\ \&\&\ A15.N >= 1\ \&\&\ needDoctorIcu1 == true\ imply\ t1d <= 240\ \&\&\ A15.D >= 1$

In order to test this property, we model a system in which there is a nurse at $NurseStation1$, a nurse in $ICU1$, a patient in $ICU1$ (location A_{15}, and a doctor at $Floor1$. The behaviors of each of them (discussed in the next section) is set to such that all five predicates are satisfied. The output is shown in Figure 5.3.

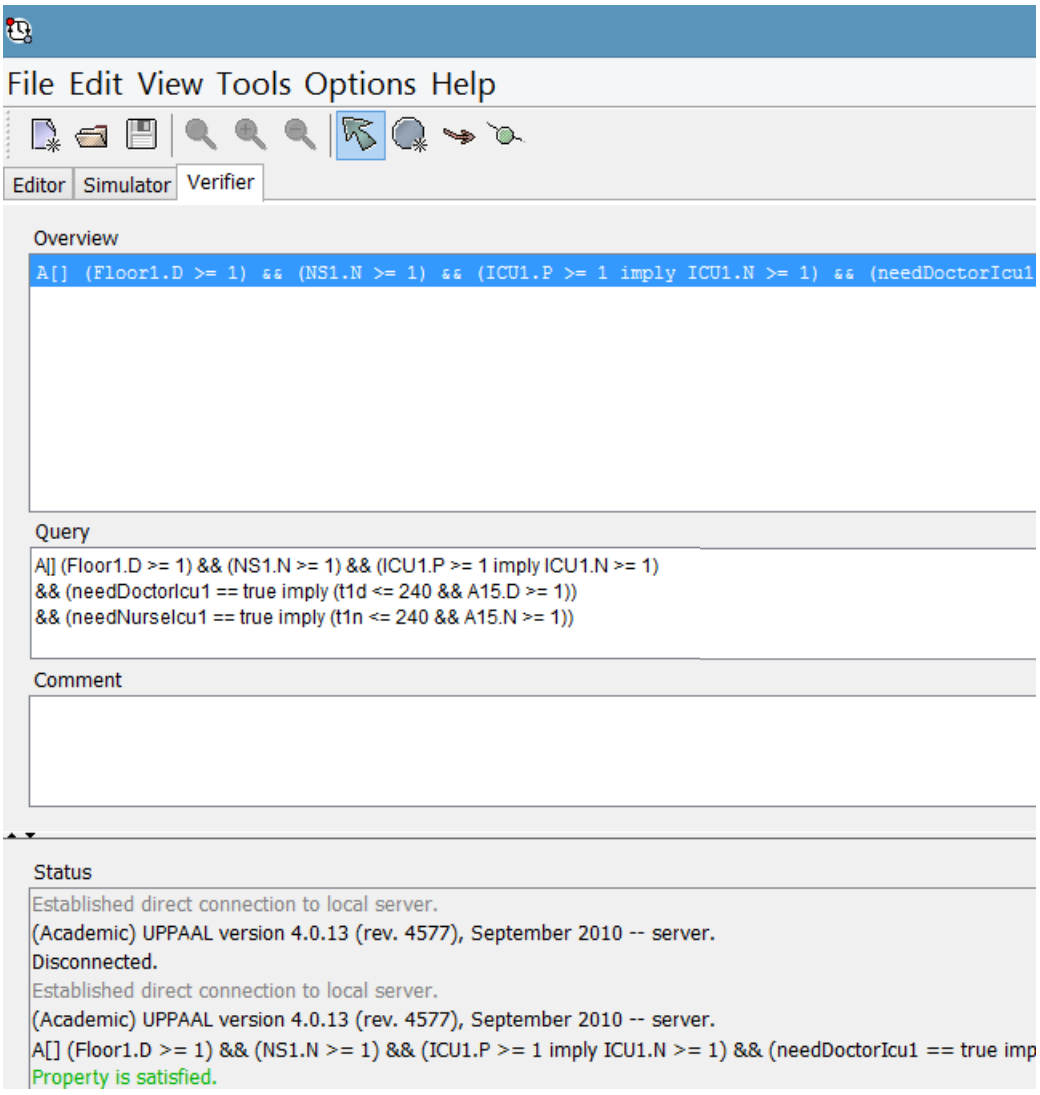


Figure 5.3: *The output of the model checker when given five strong predicates as input.*

5.5.4 Model Checking Results

In this section, we present model checking results of the CPS shown in Figure 1.2. There are four factors which influence the outcome of the model checking: temporal constraints on predicates, number of providers, number of requesters, and providers' and requester's behaviors. We model check the system under various scenarios by varying one or more of these factors.

We start with a base case in which all the predicates have string temporal requirements, and the number of providers, requesters, and their behavior is such that all the predicates are satisfied. We then increase the number of requesters, and then find that the predicates start violating. We try first three options, *i.e.*, change number of users (requestors and providers), change their behavior, and weaken the predicates, and investigate different scenarios in the following sections.

Base Case

Set of predicates assuming $t_n = t_d = 4$:

$pred_1$: $\forall ns \in NS, |ns.Nurse| \geq 1$

$pred_2$: $\forall icu \in ICU, |icu.Patient| \geq 1 \rightarrow |icu.Nurse| \geq 1$

$pred_3$: $\forall f \in Floor, |f.Doctor| \geq 1$

$pred_4$:$p \in Patient, p.location = A$

t: $p.needsNurse = true$

$\exists n \in Nurse$

$t \leq t' \leq t + 4$: $n.location = A$

$pred_5$: $p \in Patient, p.location = A$

t: $p.needsDoctor = true$

$\exists d \in Doctor$

$t \leq t' \leq t + 4$: $d.location = A$

Number of Nurses: 2

Nurse 1 Behavior: The nurse enters $NurseStation1$ at 12 PM and leaves at 8 PM. During this time, if a patient, who is not in $ICU1$ needs nurse's service, she visits the patient, provides her services, and comes back to $NurseStation1$. The nurse can sit either in area A_2 or A_3.

Nurse 2 Behavior: The nurse enters $ICU1$ at 12 PM and leaves at 8 PM. During this time, if a patient, who is in $ICU1$ needs nurse's service, she visits the patient, provides her services and comes back to her seat. We assume that she sits in area A_{15}.

Number of Doctors: 1

Doctor 1 Behavior: The doctor enters $Floor1$ at 12 PM and leaves at 8 PM. During this time, if a patient needs doctor's service, she visits the patient, provides her services, and comes back to A_3. We assume that if the doctor is not providing her services to any patient, then she comes back to area A_3.

Number of Patients: 1

Patient 1 Behavior: The patient enters $ICU1$ at 1 PM and leaves at 7 PM. During this time, he needs doctor's and nurse's services alternatively every 15 minutes. The service time $t_s = 4$, *i.e.*, a patient needs provider's services for 4 minutes. We assume that the patient's bed in $ICU1$ is located in area A_{16}.

With these as input, if we model check the system, we find that all the predicates are satisfied. The output is shown in Figure 5.3. We now increase the number of requesters and analyze the output in the following.

Case 1

In base case, if we add one more patient with the same behavior as patient 1 except that he enters $Room1$ instead of $ICU1$, we find that $pred_2$ is violated. It is bound to happen because:

- This patient needs nurse's services every 30 minutes.
- There is only one nurse available at $NurseStation1$.

- The nurse's behavior is such that when a patient not in $ICU1$ needs services, she has to visit the patient. Therefore, when the patient in $Room1$ needs nurse's services, the nurse visits him and hence $pred_2$ is violated.

Table 5.1 shows the effect on predicates with varying number of requesters and providers with the behavior same as base case except their location changes. We notice that as the number of requesters increases, the number of providers need to be increased in order for all the predicates to be satisfied.

ICU1.N	**NS1.N**	**Floor1.D**	**ICU1.P**	**Room1.P**	**Room2.P**	**Violation**
1	1	1	1	0	0	None
1	1	1	1	1	0	$pred_2$
1	2	1	1	1	0	$pred_5$
1	2	2	1	1	0	None
1	2	2	1	1	1	$pred_2$
1	3	2	1	1	1	$pred_5$
1	3	3	1	1	1	None

Table 5.1: *Various scenarios when the number of requesters an providers are increased.*

Increasing the number of providers is not the only way to make all predicates true. We may weaken one or more predicates such that all the predicates are satisfied.

Case 2

Assume that it takes maximum δ minutes for an active entity to move the largest distance at $Floor1$. If we weaken $pred_2$ such that it can remain false for $t_s + \delta$ minutes, $pred_2$ will still be satisfied, because, nurse can leave nurse station to service a patient for t_s minutes and can come back in δ minutes. If there is another request, then the nurse can service that patient. We notice that when there is one nurse at $NurseStation1$, and one patient in $Room1$, $pred_2$ is satisfied, but $pred_3$ is not satisfied, because, patient in $ICU1$ and patient in $Room1$ may need a doctor at the same time.

Now if we weaken $pred_2$ such that it can remain false for *(number of patients outside ICU1)* * $t_s + \delta$ minutes, and increase t_n and t_d to *(number of patients outside ICU1)* * $t_s + \delta$ minutes for $pred_4$ and $pred_5$ respectively, we find that all the predicates are satisfied irrespective of number of requestors. The results are shown in Table 5.2.

ICU1.N	NS1.N	Floor1.D	ICU1.P	Room1.P	Room2.P	Violation
1	1	1	1	0	0	None
1	1	1	1	1	0	None
1	2	1	1	1	0	None
1	1	2	1	1	0	None
1	1	1	1	1	1	None
1	1	2	1	1	1	None
1	2	1	1	1	1	None

Table 5.2: *Various scenarios when some of the predicates are weakened.*

We observe for $pred_2$ that the nurse $N1$ located in the ICU can not leave as long as there is a patient in the ICU. We explore the possibility of providing flexibility to $N1$ so that she can leave ICU if another nurse enters the ICU. For this, we employ CyS, such that if nurse $N2$ enters ICU, the cyber algorithm provides this state information to $N1$, allowing $N1$ to leave if required. These algorithms can be made more efficient so that they generate warnings if a predicate violation is detected. We discuss these cyber algorithms in the next section.

5.6 Predicate Detection Algorithm

The two objectives of the cyber algorithm running in CyS are:

1. generate warnings if any of the predicates becomes false, and

2. manipulate active entities' behavior by providing additional information.

Since the algorithm detects predicate violations, we also call it predicate detection algorithm. We propose both centralized and a distributed algorithms. In the following sections, we discuss these algorithms and simulation results.

5.6.1 Centralized Algorithm

In centralized algorithm, all the updates are forwarded to a central server. The server evaluates all the predicates, provide additional information to entities, and generate warnings if predicate violation is detected. When a cyber entity senses or loses a physical entity, it forwards this update to the server. The server then updates its local variables to reflect the changes. The request message by the requestors are also forwarded to the server. The server then passes it to appropriate providers.

5.6.2 Distributed Algorithm

In distributed algorithm, for each predicate, we chose a cyber entity and forward all the updates which influences the outcome of the predicate to that cyber entity. For location predicates (assuming that they involve coarse areas, such as $pred_1$ to $pred_3$), all the updates are forwarded to one of the entities monitoring an entrance area of that coarse grained area. That cyber entity is responsible for evaluating the predicate, generating warnings, and providing additional information to active entities. For service predicates, we assume that free providers are located at a particular location. The request is forwarded to one of the cyber entities monitoring that location. For example, free nurses are located at $NurseStation1$ in Figure 1.2, and the request is forwarded to either C_2 or C_3.

We simulate the two algorithms in UPPAAL and compare the results under various scenarios as discussed in the next section.

5.6.3 Simulation and Results

We measure the efficiency of these algorithms in terms of number of messages generated in the system, effect of sensing delay in detecting predicate violation, the number of times additional information is provided to the users, and the number of times the users use this additional information. The algorithms are executed for the CPS model shown in Figure 1.2. We evaluate $pred_1$ to $pred_5$ as discussed in earlier sections. We assume that the requests (to get nurse) by patients in $ICU1$ are forwarded to C_{15}, requests by patients not in $ICU1$ are forwarded to C_2, and requests to get doctors are also forwarded to C_2. For centralized algorithm, we assume that the server is connected to C_1. For distributed algorithm, we assume following:

- $pred_1$ is evaluated at C_2.
- $pred_2$ is evaluated at C_{15}.
- $pred_3$ is evaluated at C_{10} (entrance of $Floor1$).
- Requests to get the nurse are forwarded to C_2 (for patients not in $ICU1$), and C_{15} (for the patients in $ICU1$).
- Requests to get the doctor are forwarded to C_2.
- A patient needs services of a provides 10 times over a period of 5 hours.

Unless specified, the transmission delay T_d is 5ms, sensing delay S_d is 0 seconds, and it takes an active entity 3 seconds to move from a fine grained area to other. The results for the base case as discussed in Section 5.5.4 are shown in Figure 5.4

Note that number of message for $pred_3$ are more in distributed algorithm than in centralized algorithm. It is because the entrance of $Floor1$ A_{10}. Where ever the doctor goes on $Floor1$, sense and lose messages are sent to C_{10}. while in centralized algorithm, these messages are sent to the server which is connected to C_1. For this set of predicates, the average

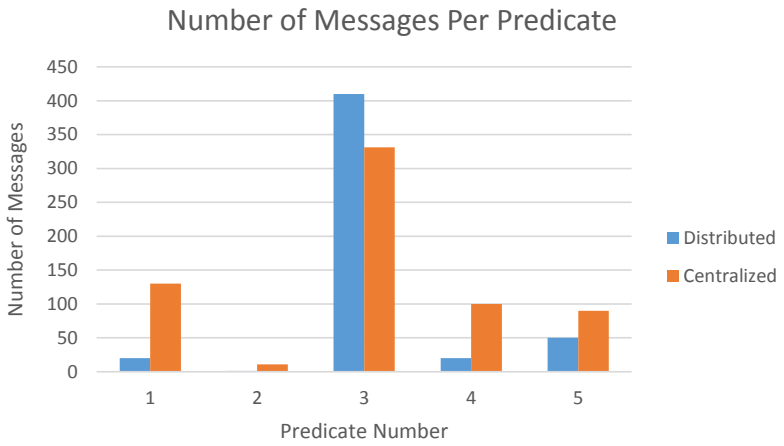


Figure 5.4: *Number of messages required to evaluate the predicates when a patient needs doctor's and nurse's services 10 times over a period of 5 hours.*

number of hops required to send a message to C_1 is less than the average number of hops required to send the message to C_{10}. These results suggest that if the predicate involves a coarser area, the more number of messages may be required in distributed algorithm.

If we change previous scenario by adding one patient in $Room1$, one more nurse at $NurseStation1$, and one more doctor to $Floor1$, we observe the results as shown in Figure 5.5. In this scenario too, all the predicates are satisfied. Instead of one patient, there are two patients in this scenario, and one should except the results for this scenario to be similar to the previous scenario, which the results prove.

For the results shown in Figure 5.4, we assume that sensing delay S_d is 0 seconds, *i.e.*, cyber entities are able to detect presence (absence) of physical entities as soon as they move inside (outside) their sensing range. If S_d is increased, one should expect false positives. However, in above two scenarios, if we increase S_d from 5ms to 3500ms, we do not see any false positive. It is because there are sufficient number of providers to service all the requesters at the same time and all the predicates are already satisfied (from model checking results). In addition, the patient expect the doctor or nurse within 4 minutes of making request, which is much larger than 3500ms and *CyS* is able to evaluate the service predicates

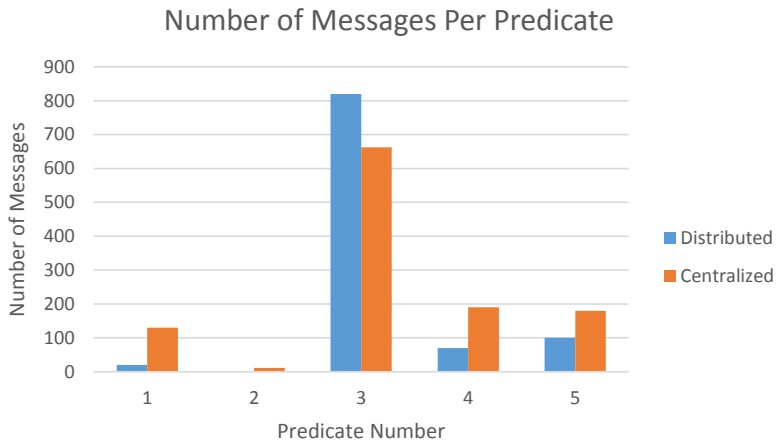


Figure 5.5: *Number of messages required to evaluate the predicates when two patients need doctor's and nurse's services 10 times over a period of 5 hours.*

in time.

Now if we simulate a scenario represented by row number 5 of Table 5.2, we obtain the number of messages per predicate as shown in Figure 5.6. It will be interesting to see the effect of increasing S_d on number of warnings generated as the number of providers are less than the number of requesters. Figure 5.7 shows that as S_d increases, the number of warnings generated increases. It happens because the in a predicate, say in $pred_4$, patient will wait for certain amount of time (*(number of patients outside ICU1)* * $t_s + \delta$) for the doctor to arrive. Two patients may may at the same time request for the doctor. The doctor reaches the second patient within time limit, but since due to large S_d, *CyS* fail to sense the doctor's presence and generates the warning.

We now consider a scenario where an extra nurse keeps moving between $NurseStation1$ and $ICU1$. We want to observe following:

- how many times additional information is provided to the nurses so that they can leave $ICU1$ without violating $pred_2$?
- how many times the information is utilized by the nurses?
- how S_d affects the accuracy of information?

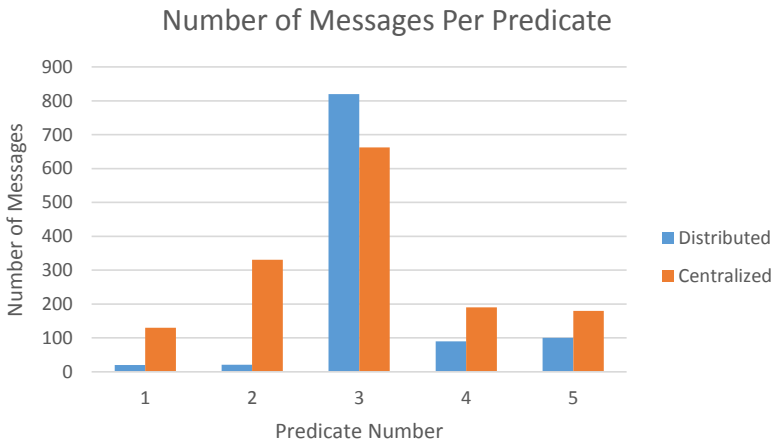


Figure 5.6: *Number of messages required to evaluate the predicates when two patients need doctor's and nurse's services 10 times over a period of 5 hours.*

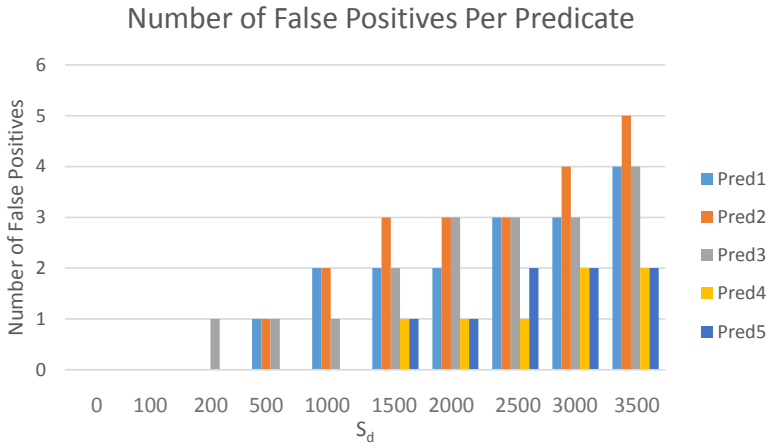


Figure 5.7: *Observation of how S_d impacts evaluation of predicates.*

We begin with assumption that there is no CyS and the $ICU1$ is big enough that the nurses can not see each other. When an additional nurse moves inside $ICU1$, the other nurse does not know about her presence. We observe that even if the other nurse want to leave $ICU1$ for some time, she can not do so, because her behavior does not allow her to leave $ICU1$. We now assume that the cyber algorithm is monitoring the physical system and evaluate above mentioned items. We also assume that the additional nurse moves in and out of $ICU1$ for 10 times. This causes CyS to provide additional information 20 times, once per entry and exit of the nurse from $ICU1$.

Figure 5.8 shows that when $S_d < 200ms$, CyS always provided accurate information, and action of the nurses did not cause $pred_2$ violation. As 23 increase S_d, we notice that number of times the wrong information provided increases, and consequently, nurses' actions cause predicate violation.

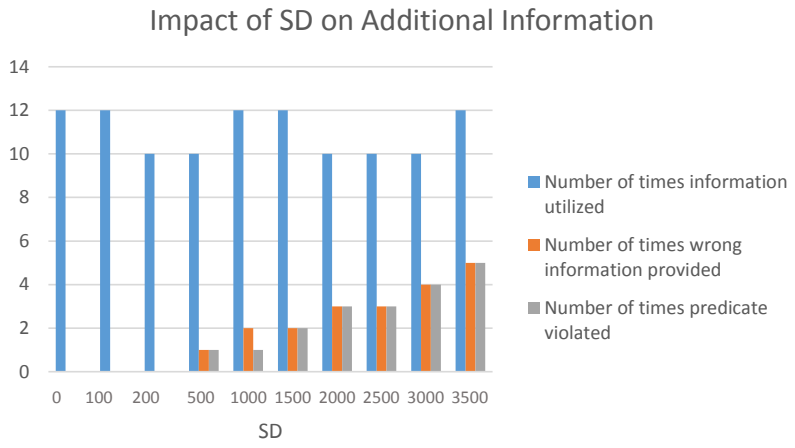


Figure 5.8: *Observation of how active entities utilize the information provided by CyS, and how S_d impacts this.*

5.7 Summary

In this chapter, we presented predicate model, model checking techniques, and predicate detection algorithms for CPSs. We studied various scenarios with the same set predicates

but varying degree of weakness, different number of providers and requestors with varying behavior. We observed that for strong predicates to be satisfied, the number of providers need to increase linearly with the increase in the number of requestors. As we weaken the predicates, less number of providers are required for all predicates to be satisfied. The result shows that in distributed algorithm, if a predicate involves a much coarser area, then the number of messages required to continuously monitor that predicate may be more. It suggests that predicates should be formulated carefully. Model checking of a CPS is useful in the sense that the results can be used to determine the number of providers and their schedule in the system. It can also be used to find flaws in the given set of predicates such as no matter what, one or more predicates are never satisfied.

Chapter 6

Global State Recording

In the previous chapter, we observed that as we increase sensing delay (S_d), the number of false positives also increases. It is because *CyS* fails to reflect the current state of *PhyS* in timely manner. In this chapter, we present an approach to record global state, which helps *CyS* to maintain more accurate state of *PhyS*. We have not evaluated it experimentally, however, we believe that it constructs a more accurate state of *PhyS* as compared the state which we obtain in its absence.

6.1 Introduction

There are various domains in which constructing global state of *PhyS* can be useful. [31] discusses a context aware workflow in a smart factory which requires to collect information about the usage time of a tool. [35] proposes continuous monitoring of pervasive systems through a series of snapshot queries. They discuss the applications of their approach for a smart construction site.

The key idea in such applications as discussed above is to continuously collect the state of *PhyS* and exploit it to solve underlying problem. This problem has been abstracted as global state recording problem TDSs [55, 56, 61]. In a TDS, the global state is defined as collection

of local states of all processes and local states of all communication channels. However, in a CPS, the global state is defined over *PhyS* and is recorded with the help of *CyS*. As discussed in Section 2.2.3, due to S_d and lost sensor data, *CyS* may have an inaccurate state of *PhyS*. We have shown in the previous chapter that these factors significantly affect *CyS*'s capability to maintain an accurate state of *PhyS*. In the rest of the chapter, we discuss the background of global state recording algorithms, and an approach for *CyS* to maintain a more accurate state of *PhyS*. We also discuss why it is not possible for *CyS* to maintain 100% accurate state of *PhyS*.

6.2 Background and Related Work

Global state recording in TDSs on the fly is an important paradigm when one wants to analyze and detect properties associated with the system such as termination detection and deadlock detection. This problem becomes non-trivial to solve in the absence of globally shared memory, global clock and unpredictable message delays in a distributed system. A TDS is modeled as a set of processes $P_1, \ldots, P_n$ connected via communication channels. C_{ij} denotes a communication channel between P_i and P_j, SC_{ij} denotes local state of C_{ij} and SP_i denotes local state of P_i. The actions performed by a process are modeled as three types of events: internal events, the message send event, and the message receive event. For a message m_{ij}, which is sent by process P_i to process P_j, $send(m_{ij})$ denotes its send event and $recv(m_{ij})$ denotes its receive event. At any instant, SP_i is the sequence of all the events executed by P_i till that instant. For an event e and a process P_i, $e \in SP_i$ *iff* e belongs to the sequence of events that have taken process P_i from its initial state to state SP_i; and $e \notin SP_i$ *iff* e does not belong to the sequence of events that have taken process P_i from its initial state to state SP_i. A message m_{ij} is in channel C_{ij} iff $send(m_{ij}) \in SP_i \wedge recv(m_{ij}) \notin SP_j$.

Global state GS of a TDS is defined as, $GS = \{\bigcup_i SP_i, \bigcup_{i,j} SC_{ij}\}$. A global state GS is consistent *iff* it satisfies the following two conditions:

C1: $send(m_{ij}) \in SP_i \Rightarrow m_{ij} \in SC_{ij} \oplus recv(m_{ij}) \in SP_j$. ($\oplus$ is Ex-OR operator.)

C2: $send(m_{ij}) \notin SP_i \Rightarrow m_{ij} \notin SC_{ij} \wedge recv(m_{ij}) \notin SP_j$.

Based on this model, Chandy and Lamport [55] presented a global state recording algorithm for FIFO networks as shown in Algorithm 6.1. This is the baseline algorithm for global state recording algorithms developed by other researchers. The Chandy-Lamport algorithm uses a control message, called a *marker*, which separates messages in the channels into those to be recorded in the global state from those not to be recorded in the global state.

The process which wants to record the global state initiates the algorithm by executing the "Marker Sending Rule". By executing "Marker Sending Rule" a process records its local state and sends a *marker* on each outgoing channel. On receiving a marker, a process executes the "Marker Receiving Rule". If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and records its local state by executing the "Marker Sending Rule". The algorithm terminates after each process has received a *marker* on all of its incoming channels. The pseudo code of the algorithm is given below.

Algorithm 6.1 Chandy and Lamport's Algorithm

```
1: Marker Sending Rule for process i
2:     1. Process i records its state.
3:     2. For each outgoing channel C on which a marker
4:        has not been sent, i sends a marker along C before
5:        i sends further messages along C.
6: Marker Receiving Rule for process j
7:     On receiving a marker along channel C:
8:     if j has not recorded its state then
9:         Record the state of C as the empty set
10:        Follow the "Marker Sending Rule"
11:    else
12:        Record the state of C as the set of messages
13:        received along C after j's state was recorded
14:        and before j received the marker along C.
```

Based on Chandy-Lamport's algorithm, [56, 61] proposed global state recording algorithms for FIFO systems. Spezialetti and Kearns [56] optimized the Chandy-Lamport algorithm for concurrent initiators. A process records its local state only once, irrespective of the number of concurrent initiators, and the global state is sent only to initiators. Venkatesan [61] proposed incremental global state recording algorithm when the application requires to record the global state repeatedly. This can be achieved by repeatedly executing Chandy-Lamport algorithm, however Venkatesan [61] optimized it by recording an incremental global state since the most recent global state was recorded and combine it with the most recent global state to obtain the latest global state of the system.

Global state recording in non-FIFO systems is complicated because a marker cannot be used to separate messages into those to be recorded in the global state from those not to be recorded in the global state. For such systems, [62–64] proposed algorithms which ensure that the recorded state satisfies condition **C2**.

[37] is a step towards global state recording in pervasive systems and CPSs. It presents two algorithms for event ordering in pervasive sensor networks. The first algorithm treats the entire network as a single graph, whereas the second algorithm operates in a hierarchical manner by sub dividing the network into smaller groups. Similar to the first algorithm in which a sink node receives all observable events, our approach has a global observer which observes all observable events as soon as they occur in CyS. We define a correspond function which is similar to happens before relation of [37].

[35] presents algorithms that perform continuous monitoring of physical phenomena, and provide constructs to formulate continuous queries. It presents a framework that uses statistical techniques for inferring missing and uncertain data. It also discusses various techniques to understand the degree of confidence an application can have in that inferred information. Similar to [35], our approach also has a notion of inferring sensor values, however, it is based on our proposed correspond function, and not on statistical inference techniques of [35]. We also propose a confidence function which implies the degree of the

accuracy of location of physical entities in the system. The downside of [37] and [35] are that they do not accommodate a well defined spatial model and user behavior, which are integral part of a CPS.

6.3 Global State in CPS

We have discussed in Section 3.2.2 that the state of a fine grained area A is represented using variables $A.ae$ and $A.re$; and state of a coarse grained area A is represented using $A.ae = \bigcup_i A_i.ae$ and $A.rs = \bigcup_i A_i.rs$ such that $A \mathbb{C} A_i$. We also discussed in Section 3.2.3 that a cyber entity monitoring a fine grained area A maintains its own set of variables $A.ae$ and $A.rs$. These variables change as the state of A changes due to various actions performed by active entities. The global state of the *PhyS* should consists of all the physical entities present in the system. The physical entities which are present in fine grained areas are included in the state using the two variables discussed, however, the entities which are in transit from an area to other area are not included. These entities are located in one of the reachability edges. In order to model this, we introduce two more variables $R_{ij}.ae$ and $R_{ij}.rs$, which represent the set of active entities and resources respectively in transit from A_i to A_j or A_j to A_i. Thus, the global state of a *PhyS* is defined as $GS_P = \{\bigcup_i A_i.ae, \bigcup_i A_i.rs, \bigcup_{i,j} R_{ij}.ae, \bigcup_{i,j} R_{ij}.rs\}$

Since active entities and resources are physical entities, for clarity of the expression, we use a single variable $A.pe$ and $R_{ij}.pe$ to include both. Therefore, $GS_P = \{\bigcup_i A_i.pe, \bigcup_{i,j} R_{ij}.pe\}$ For GS_P to be consistently reflected in *CyS*, following two requirements must met:

- *CyS* should be able to sense reachability edges, and
- the sensing delay should be 0, so that *CyS* updates its variables as soon as corresponding variables in *PhyS* are updated.

The first condition states that each and every part of the physical system should be in sensing range of a cyber entity. This is very difficult to achieve and not cost effective. Due to

hardware limitations such as sensing delays, and signal interferences, the second condition is also hard to achieve. Because these requirements cannot be met, there are two possible inconsistencies which in the state of $PhyS$ as maintained by CyS:

- A physical entity is not sensed by any of the cyber entities at all, because that entity might be in a reachability edge.
- A physical entity is included in more than one cyber entities' corresponding $A.pe$ variables. For example, in Figure 1.2, nurse N_6 may move from A_3 to A_2. C_2 may start sensing N_6 and update its $A_2.ae$ variable as soon as N_6 enters A_2. However, it may be possible that due to S_d or missing sensor values, C_3 still hasn't updated $A_3.ae$ to remove N_6 from it. This causes N_6 to be included twice in the global state.

As discussed in Section 3.2.3, $Move$ action by a physical entity U causes $Sense(U)$ and $Lose(U)$ events to be generated in cyber entities, and as a consequence, they update corresponding $A.pe$ variables. Our approach is based on the following assumptions:

- **A1**: A global observer exists which observes all the events $Sense$ and $Lose$ events generated in all the cyber entities as soon as the occur. It also maintains its own clock and timestamps events as it observes them.
- **A2**: For three areas A_i, A_j, and A_k, R_{ij} and R_{jk} exist, and if a physical entity U moves from A_i to A_k via A_j, then C_j will definitely generate a $Sense(U)$ and $Lose(U)$ events. It eliminates the possibility of CyS reflecting that U moved from an area to another such that a reachability edge does not exist between them. For example, if in Figure 1.2, N_6 moves from A_3 to A_{16} via A_{15}, then C_{15} does generate $Sense(U)$ event.
- **A3**: A physical entity U can move out of an area A_i, and without entering any other area and may move back into A_i. We assume that a self reachability edge from A_i to A_i (R_{ii}) exists.

- **A4**: The direction of physical entities' movement can not be sensed. For example, in Figure 1.2, if nurse N_6 moves out of A_3, and currently in transit to A_{15}, then *CyS* has no way to figure out that it is actually moving towards A_{15}. For *CyS*, she may be moving towards either of A_2, A_5 or A_{15}; or she may move back to A_3. Therefore, *CyS* may not be able to maintain $R_{ij}.pe$ variables for individual reachability edges. Therefore, when a physical entity U moves of of an area A_i, *CyS* maintains that U belongs to one of the reachability edges incident on A_i. To model this, we assume that *CyS* maintains a variable $RS_i.pe = \bigcup_{i,j} R_{ij}.pe$, and updates it whenever a *Lose* event is generated in C_i

Based on above assumptions, we define following terms.

Definition 6.1. *(Global State of CPS): The global state GS_C of CPS is GS_P as maintained in* CyS. *It is defined as $GS_C = \{\bigcup_i A_i.pe, \bigcup_i RS_i.pe\}$*

Definition 6.2. *(Ground State of CPS): Ground State [65] at time t (GR^t) is GS^t such that at time t, $\forall i RS_i.pe = \{\Phi\}$, i.e. there are no physical entities in transit in any of the reachability edges.*

Definition 6.3. *(Computation): Let $lose_{i,U,t}$ and $sense_{i,U,t}$ denote Lose and Sense events generated when physical entity U moves out or inside an area A_i at time t. Computation Π is defined as the set of all $lose_{i,U,t}$ and $sense_{i,U,t}$ events generated in the system.*

We allow an insert operation on Π which inserts an event in Π according to ascending order of time of its occurrence.

Definition 6.4. *(Correspond Function): Correspond function $\mathcal{F}$ is a one-to-one function from Π to Π such that*

- $\mathcal{F}(sense_{i,U,0}) = sense_{i,U,0}$
- *For $s, t > 0$ and $t > s$, $\mathcal{F}(sense_{i,U,t}) = lose_{j,U,s}$ and R_{ij} exists, and $\nexists sense_{k,U,u}, lose_{l,U,v} \in \Pi$ such that $s < u < t$ and $s < v < t$.*

Definition 6.5. *(Consistent Computation): Computation* Π *is consistent if for every* $sense_{i,U,t}$, $t > 0$, $\mathcal{F}(sense_{i,U,t}) \in \Pi$.

Definition 6.6. *(Latest event for a physical entity): Latest event for a physical entity* U *is either a* $sense_{i,U,t} \in \Pi$ *or* $lose_{i,U,t} \in \Pi$ *such that for* $t_1 > t$, $sense_{k,U,t_1} \notin \Pi$ *or* $lose_{j,U,t_1} \notin \Pi$.

Lemma 2. *If* $\mathcal{F}(sense_{i,U,t}) \notin \Pi$, *then the latest event for* U *in* Π *will be* $sense_{j,U,t_1}$, $t > t_1$ *and* R_{ij} *exists.*

Proof. We will prove it by contradiction. Assume that when $\mathcal{F}(sense_{i,U,t}) \notin \Pi$, then the latest event for U in Π is $lose_{j,U,t_1}$, $t > t_1$ and R_{ij} exists. From the definition of $\mathcal{F}$ and latest event, $\mathcal{F}(sense_{i,U,t}) = lose_{j,U,t_1}$, which is contradiction. Hence, when $\mathcal{F}(sense_{i,U,t}) \notin \Pi$, then the latest event for U in Π will be $sense_{j,U,t_1}$, $t > t_1$ and R_{ij} exists. □

We assume that CPS is initialized in GR^0, *i.e.*, at $t = 0$, for each physical entity U located in A_i, Π contains $sense_{i,U,0}$. From Definition 6.5, Π at $t = 0$ is consistent. At later point of time, due to S_d, it is possible that when a physical entity U moves out of A_i, and moves in A_j, a *lose* event hasn't been generated in C_i, while a *sense* event has been generated in C_j. This causes Π to reflect U to be sensed by both C_i and C_j, resulting in an inconsistent Π. Our aim is for the global observer to always compute a consistent Π. The observer computes a consistent Π using Algorithm 6.2

Theorem 3. *Algorithm 6.2 always computes consistent* Π.

Proof. From our assumption that CPS is initialized in GR^0, at time $t = 0$ Π is consistent. Assume that for $t > 0$, Π is consistent. We will prove that for $t_1 > t$, Π is consistent. At $t_1 > t$, there are two possibilities:

(Observer receives $lose_{i,U,t_1}$ event before it receives another $sense_{j,U,t_1}$ event for a physical entity U): In this case, Line 14 will be executed and the $lose_{i,U,t_1}$ will be inserted to Π. Since old value of Π was consistent, adding a $lose_{i,U,t_1}$ event will not cause it to be inconsistent (from Definition 6.5).

Algorithm 6.2 Algorithm to calculate consistent Π

1: **On observing** $sense_{i,U,t}$
2: Check if $\mathcal{F}(sense_{i,U,t}) \in \Pi$
3: if yes then
4: Insert $sense_{i,U,t}$ to Π
5: else
6: Search for latest event for U in Π
7: Assume that it is generated at C_j at time t_1
8: Chose t_2 such that $t_1 < t_2 < t$, and insert $lose_{j,U,t_2}$ to Π
9: Insert $lose_{j,U,0}$ to $IgnoreLose$ set
10: **On observing** $lose_{i,U,t}$ for process j
11: If $lose_{i,U,0} \in IgnoreLose$ then
12: Remove $lose_{i,U,0}$ from $IgnoreLose$
13: else
14: Insert $lose_{i,U,t}$ to Π

(Observer receives a $sense_{j,U,t_1}$ event without receiving a $lose_{i,U,t_1}$ event for a physical entity U): In this case, Line 6-9 will be executed. From Lemma 2, Line 6-7 finds a latest $sense_{j,U,t_2}$, $t_2 < t_1$ for U. Line 8 inserts a $lose_{j,U,t_3}$ to Π such that $t_2 < t_3 < t_1$. From Definition 6.4, $\mathcal{F}(sense_{j,U,t_1}) = lose_{j,U,t_3}$, and hence from Definition 6.5, Π is consistent. Line 9 simply insert a $lose_{j,U,0}$ event to $IgnoreLose$ set so that when the actual $lose_{j,U,t}$ event is received, it is ignored and removed from $IgnoreLose$ set (Line 11-12). □

Once the observer has a consistent Π, the global state at a given point of time can be constructed using Algorithm 6.3.

Algorithm 6.3 Algorithm for constructing GS_C from Π

1: **On observing** $sense_{i,U,t}$
2: For each physical entity U find latest event in Π
3: if the latest event is $sense_{i,U,t}$ then
4: add U to $A_i.pe$
5: else
6: add U to $RS_i.pe$

Since there is exactly one latest event corresponding to each physical entity in Π, Algorithm 6.3 ensures that each physical entity is included exactly once in the global state.

However, the constructed global state may not reflect the current v alues of $A.pe$ variables, as Theorem 4 proves.

Theorem 4. *If $S_d > 0$, Π at time t_1 reflects the state of* PhyS *at time $t, t_1 - S_d < t < t_1$.*

Proof. Assume that a set $\mathcal{S} = \{C_1, \ldots, C_n\}$ of cyber entities is monitoring a *PhyS*. Also assume that current computation is consistent. Now suppose that a physical entity U moves out of A_i at time t, for which a $lose_{i,U,t_1}$ event will be generated in C_i, and subsequently added to Π. If C_i reads the sensor values just before U moves out of A_i, then C_i will read next sensor values after S_d time unit. Thus, $t < t_1 < t + S_d$. In other words, $t_1 - S_d < t < t_1$. Therefore, Π at a given time t_1 reflects the past state of *PhyS* at time t such that $t_1 - S_d < t < t_1$. □

In order to accommodate this temporal inconsistency, we propose a confidence function which is calculated continuously by the observer while it calculates Π. Confidence function associates a positive real number in the range $[0, 1]$ with each physical entity U located in a fine grained area A_i. This number is represented as $c(U, A_i)$, and more towards 1 it is, more are the chances that U is located in A_i. We discuss it in detail in the following.

Confidence Function

Consider an active entity U enters a fine grained area A_i. As soon as C_i starts sensing its presence at time t (*i.e.*, $sense_{i,U,t}$ event is generated), $c(U, A_i)$ is assigned a value σ slightly greater than 0, but much less than 1. From Theorem 4, the move action by U which caused $sense_{i,U,t}$ event in C_i might have occurred in *PhyS* at most $t - S_d$ time ago. By the time the observer observes $sense_{i,U,t}$ event, there are chances that U might have left A_i. If this is the case, the observer will observe $lose_{i,U,t_1}$, $t < t_1 \leq t + S_d$, which from Theorem 4 might have occurred in *PhyS* at most $t_1 - S_d$ time ago. Assume that the observer does not observe $lose_{i,U,t_1}$ event up to $t + S_d$. It implies U did not move out of A_i at least up to time t. The confidence that U is located in A_i increases. Thus, after S_d time units, $c(U, A_i)$ can be incremented by a fraction ϕ. The process of incrementing $c(U, A_i)$ continues until $c(U, A_i)$ becomes 1 or the observer observes a $lose_{i,U,t}$ event. In this case, $c(U, A_i)$ is set to 0.

If the observer maintains confidence of each physical entity, then it can be used to determine the accuracy of the current location of physical entities in the system. We have not evaluated it empirically, however, we believe that it can significantly improve inferring the location of physical entities.

In the following, we discuss few applications of global state recording in CPSs.

6.4 Applications

Global state recording in CPS can be exploited to solve various other problems such as predicate detection, continuous query, and discrete query.

6.4.1 Predicate Detection

We have discussed in the previous chapter that due to larger S_d, message loss, missed sensor values, or physical entities being located in reachability edges, predicate detection algorithm generates incorrect warnings. It can evaluate predicates more accurately if *CyS* correctly records the state of *PhyS*. For example, when nurse N_6 in Figure 1.2 is in transit to A_2, and given that reachability edges are not monitored by *CyS*, N_6's presence does not get reflected in *CyS*. In this case, if there is a predicate, say, N_6 *should always be in* $NurseStation1$, predicate detection algorithm will generate a warning. Given GS_C, predicate detection algorithm utilizes the consistent computation Π and waits until a $sense_{i,N_6,t}$ is inserted in Π. When predicate detection algorithm finds that a $sense_{i,N_6,t}$ event has been inserted in Π, then it can evaluate the predicate as follows: if i corresponds to an area outside of $NurseStation1$, then generate warning, otherwise the predicate is satisfied.

6.4.2 Continuous query

It is sometimes required to assess the usage pattern of a resource during a certain time interval to more efficiently ensuring its availability during peak usage time. Given GS_C,

we can find a subset of Π such that it contains all the events in a time interval, and have continuous query algorithm to analyze it and answer the query.

6.4.3 Discrete Query

: One may also want to query *CyS* about the current state of *PhyS*. For example, find the number of nurses currently available in maternity ward is a discrete query. If global state recording algorithm is in place, we can find subset of Π containing all the events which have happened in the cyber entities monitoring fine grained areas contained in maternity ward. Discrete query algorithm can use this subset to answer the query.

6.5 Summary

In this chapter, we discussed the problem of global state recording in CPSs. We presented an approach to address the problem, and proved its correctness. We also discussed the factors due to which it is not possible for *CyS* to maintain a 100% accurate global state of *PhyS*. Further work is needed to design the algorithms for the global observer. It could be a centralized or a distributed algorithm, and these algorithms must also contend with message transmission delays.

Chapter 7

Conclusion and Future Work

The work in this thesis can be divided into four aspects of CPSs: modeling, mutual exclusion algorithm, predicate detection algorithm and model checking, and global state recording algorithm. This chapter, we conclude by summarizing each of these aspects and highlighting the work that needs to be done to improve them.

Graph based models with various assumptions related to message transmission and processing times have provided a strong foundation to study distributed algorithms in a TDS. The work in this thesis provides a step towards studying similar algorithms for a CPS. We proposed a graph based CPS model which accommodates spatial model of CPS, user behavior, and interactions between the users and the cyber system. The CPS solutions which we obtain based on this model consist of two parts: user behavior specification, and cyber algorithms running in the cyber system. Each of them can individually be changed to obtain a new CPS solution. For example, in one solution, users may behave in a disciplined manner by always following cyber system's instructions. In another behavior, apart from following cyber system's instructions, they may act on their own. The goal of designing and simulating different CPS solutions is to identify scenarios in which a particular solution would perform the best. We also present CPSML, a language to specify our proposed CPS model in a programmatic fashion. We plan to improve CPS model and CPSML in the

future as follows:

- We will extend the model to allow one to associate different properties with reachability edges to enable search for specific paths between two locations. For example, one may want to search a path between two locations such that the path involves no staircase.
- Except for a simple scenario in Chapter 4, we have not studied user-to-user interactions in detail. We will formulate constructs to model these types of interactions in our future work.
- We plan to build a graphical IDE for CPSML which will allow designers to specify CPS models by dragging and dropping various components.
- We plan to construct a tool to generate simulator code by compiling CPSML code so that the model can be simulated on the fly.

For the mutual exclusion problem, we proposed a centralized solution and two distributed solutions. We simulated these solutions using OMNeT++ simulation engine for different user behaviors and varying number of resources and users. In all scenarios, we used mxn grid to simulate physical areas. We found that there is no best solution to this problem. For certain scenarios, a particular solution could be the best, but the same solution may be inefficient for a different scenario. For example, the results show that when fewer resources are present, it might be best to rely completely on CyS; otherwise, participation of users in locating resource can improve performance. The following issues will be addressed as future work to improve our approach:

- We simulated the mutual exclusion algorithms assuming that spatial model is a grid of size mxn. We plan to extend to simulation results for spatial model with different topologies.
- We plan to device solutions for more complex mutual exclusion problems in which users may request for more than one resource.

- Finally, in the $SPRA$ algorithm, identifying mechanisms via which existing tree information could be utilized in creating on-demand paths to reduce number of messages is a subject of future research.

For predicate detection and model checking, we used the UPPAAL model checker as the simulation tool. For this aspect, we considered active entities as requestor or providers, where requestors request for obtaining services from providers. We presented a centralized and a distributed solution to solve the predicate detection problem, and simulated them for varying the number of requestors and providers, by varying the user behavior, and by weaking the predicates. The results suggest that for weaker predicates, less number of providers are required to satisfy requests. The model checking technique we presented provides useful results in terms of designing predicates and deciding on the number of providers. It requires a spatial model, a set of predicates, the number of requestors, providers and their behaviors as input, and the model checking identifies whether this combination will cause any of the predicates to be violated. In the future, we plan to study the following for CPS predicates:

- We plan to study more complex predicates which consist of multiple components separated by and/or operators. For example, given three simple predicates a, b and c, detect violation of *a and b and c* or detect violation of *a or b and c*.

- We plan to build an automated tool to generate an UPPAAL model out of given CPSML program.

For global state recording, we proposed an approach in which a global observer observes all *sense* and *lose* events occurring in the cyber entities. Based on the events it observes, it constructs a sequence of events in which corresponding to each *sense* event, there is exactly one *lose* event. We construct the global state of $PhyS$ based on this sequence. We also present a confidence function which indicates the degree of accuracy of physical entities'

current location. In the future, we plan to design algorithms for the global observer, which could be centralized or distributed.

Bibliography

[1] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, September 1965.

[2] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1: 115–138, 1971.

[3] N.A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, 1980.

[4] J. H. Reif and P.G. Spirakis. Real time resource allocation in distributed systems. In *Proceedings of ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 84–94, 1982.

[5] S. Bulgannawar and N.H. Vaidya. A distributed k-mutual exclusion algorithm. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 153–160, 1995.

[6] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park. A token based distributed k mutual exclusion algorithm. In *IEEE Proceedings of the Symposium on Parallel and Distributed Processing*, pages 408–411, December 1992.

[7] P.K. Srimani and R.L. Reddy. Another distributed algorithm for multiple entries to a critical section. In *Information Processing Letters*, volume 41, pages 51–57, January 1992.

[8] J.E. Walter, G. Cao, and M. Mohanty. A k-mutual exclusion algorithm for wireless ad

hoc networks. In *Proceedings of the First Annual Workshop on Principles of Mobile Computing*, 2001.

[9] K. Raymond. A distributed algorithm for multiple entries to a critical section. In *Information Processing Letters*, volume 30, pages 189–193, February 1989.

[10] Bhopal disaster, Accessed on Nov 9 2013. URL http://en.wikipedia.org/wiki/Bhopal_disaster.

[11] Suroop Chandran and J. Joshi. Lot-rbac: A location and time-based rbac model. In Anne Ngu, Masaru Kitsuregawa, Erich Neuhold, Jen-Yao Chung, and Quan Sheng, editors, *Web Information Systems Engineering WISE 2005*, volume 3806 of *Lecture Notes in Computer Science*, pages 361–375. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30017-5. URL http://dx.doi.org/10.1007/11581062_27.

[12] D. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1983.

[13] Ying Tan, Mehmet C. Vuran, and Steve Goddard. Event model for cyber-physical systems. In *Proceedings of the 2nd International Workshop on Cyber-Physical Systems*, 2009.

[14] Y.P. Fallah and R. Sengupta. A cyber-physical systems approach to the design of vehicle safety networks. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 324 –329, june 2012.

[15] Fengzhong Qu, Fei-Yue Wang, and Liuqing Yang. Intelligent transportation spaces: vehicles, traffic, communications, and beyond. *Communications Magazine, IEEE*, 48 (11):136 –142, november 2010.

[16] Xu Li, Xuegang Yu, A. Wagh, and Chunming Qiao. Human factors-aware service scheduling in vehicular cyber-physical systems. In *INFOCOM, 2011 Proceedings IEEE*, pages 2174 –2182, april 2011.

[17] Jing Lin, S. Sedigh, and A. Miller. Towards integrated simulation of cyber-physical systems: A case study on intelligent water distribution. In *Dependable, Autonomic and Secure Computing, 2009. DASC '09. Eighth IEEE International Conference on*, pages 690 –695, dec. 2009.

[18] M.D. Ilic, Le Xie, U.A. Khan, and J.M.F. Moura. Modeling future cyber-physical energy systems. In *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, july 2008.

[19] Yan Sun, B. McMillin, Xiaoqing Liu, and D. Cape. Verifying noninterference in a cyber-physical system the advanced electric power grid. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 363 –369, oct. 2007.

[20] N. Correll, N. Arechiga, A. Bolger, M. Bollini, B. Charrow, A. Clayton, F. Dominguez, K. Donahue, S. Dyar, L. Johnson, H. Liu, A. Patrikalakis, T. Robertson, J. Smith, D. Soltero, M. Tanner, L. White, and D. Rus. Building a distributed robot garden. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1509 –1516, oct. 2009.

[21] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[22] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.

[23] Walid Taha and Roland Philippsen. Modeling basic aspects of cyber-physical systems. *CoRR*, abs/1303.2792, 2013.

[24] A.D. Kshemkalyani, A.A. Khokhar, and Min Shen. Execution and time models for pervasive sensor networks. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 639 –647, may 2011.

[25] Sumeet Gujrati and Gurdip Singh. Mutual exclusion in cyber-physical systems. In *1st International Conference on Sensor Networks*, February 2012.

[26] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. Geo-rbac: a spatially aware rbac. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, SACMAT '05, pages 29–37, New York, NY, USA, 2005. ACM.

[27] Ajith K. Narayanan. Realms and states: a framework for location aware mobile computing. In *Proceedings of the 1st international workshop on Mobile commerce*, WMC '01, pages 48–54, New York, NY, USA, 2001. ACM.

[28] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *IEEE Trans. Knowl. Data Eng*, 14(4):881–901, 2002.

[29] Gerard J Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[30] Promela language reference, Accessed on Nov 9 2013. URL http://spinroot.com/spin/Man/promela.html.

[31] Matthias Wieland, Oliver Kopp1, Daniela Nicklas, and Frank Leymann. Towards context-aware workflows. In *Proceedings of the CAISE07 Workshops and Doctoral Consortium*, 2007.

[32] Jatuporn Chinrungrueng, Udomporn Sunantachaikul, and Satien Triamlumlerd. Smart parking: an application of optical wireless sensor network. In *Proceedings of the 2007 International Symposium on Applications and the Internet Workshops*, 2007.

[33] W. Steven Conner, John Heidemann, Lakshman Krishnamurthy, Xi Wang, and Mark Yarvis. Workplace applications of sensor networks. In *USC/ISI Technical Report ISI-TR-2004-591*, 2004.

[34] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. Smartcis: Integrating digital and physical environments. *SIGMOD Record*, March 2010.

[35] Vasanth Rajamani and Christine Julien. Blurring snapshots: Temporal inference of missing and uncertain data. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, 2010.

[36] K. Romer. Temporal message ordering in wireless sensor networks. In *Annual Mediterranean Ad Hoc Networking Workshop*, 2003.

[37] L. Kaveti, S. Pulluri, and Gurdip Singh. Event ordering in pervasive sensor networks. In *5th IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing*, March 2009.

[38] S. Bapat and A. Arora. Message efficient termination detection in wireless sensor networks. In *Proceedings of the 2008 INFOCOM Workshops*, 2008.

[39] H. Kurian, A. Rakshit, and Gurdip Singh. Detecting termination in pervasive sensor networks. In *9th IEEE International Symposium on Asynchronous Decentralized Systems*, March 2009.

[40] A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 2001.

[41] Omnet++ home page, Accessed on Nov 9 2013. URL http://http://www.omnetpp.org/home/what-is-omnet.

[42] Google maps, Accessed on Nov 9 2013. URL http://http://en.wikipedia.org/wiki/Google_Maps.

[43] Karl Wessel, Michael Swigulski, , Andreas Kpke, and Daniel Willkomm. Mixim: the physical layer an architecture overview. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[44] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[45] Alexandre David, Kim Guldstrand Larsen, Gerd Behrmann, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. 2006. URL http://vbn.aau.dk/research/(5147180).

[46] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0. 2006. URL http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[47] G.R. Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, 2002.

[48] Ying Tan, Mehmet C. Vuran, Steve Goddard, Yue Yu, Miao Song, and Shangping Ren. A concept lattice-based event model for cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 50–60, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0066-7. doi: http://doi.acm.org/10.1145/1795194.1795202. URL http://doi.acm.org/10.1145/1795194.1795202.

[49] Jing Lin, S. Sedigh, and A.R. Hurson. An agent-based approach to reconciling data heterogeneity in cyber-physical systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 93 –103, may 2011. doi: 10.1109/IPDPS.2011.130.

[50] V.K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *Parallel and Distributed Systems, IEEE Transactions on*, 5(3):299 –307, mar 1994.

[51] V.K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *Parallel and Distributed Systems, IEEE Transactions on*, 7(12):1323 –1333, dec 1996.

[52] Claudio A. Ardagna, Marco Cremonini, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Supporting location-based conditions in access control policies. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, ASIACCS '06, pages 212–222, New York, NY, USA, 2006. ACM. ISBN 1-59593-272-0. doi: 10.1145/1128817.1128850. URL http://doi.acm.org/10.1145/1128817.1128850.

[53] Ajay D. Kshemkalyani. Immediate detection of predicates in pervasive environments. *Journal of Parallel and Distributed Computing*, 72(2):219 – 230, 2012.

[54] Luc Bouge. Repeated snapshots in distributed systems with synchronous communications and their implementation in csp. *Theoretical Computer Science*, 49(23):145 – 169, 1987.

[55] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[56] M. Spezialetti and P. Keams. Efficient distributed snapshots. In *Proceedings of 6th International Conference on Distributed Computing Systems*, 1986.

[57] Wen Yao, Chao-Hsien Chu, and Zang Li. Leveraging complex event processing for smart hospitals using rfid. *Journal of Network and Computer Applications*, 34(3):799 – 810, 2011.

[58] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[59] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, Mass., 1979.

[60] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[61] S. Venkatesan. Message-optimal incremental snapshots. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 53 –60, jun 1989.

[62] Jean-Michel Helary. Observing global states of asynchronous distributed applications. In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 124–135. Springer Berlin / Heidelberg, 1989.

[63] H. F. Li, T. Radhakrishnan, and K. Venkatesh. Global state detection in non-fifo networks. In *Proceedings of 7th International Conference on Distribured Compuring Systems*, 1987.

[64] F. Mattem. Efficient algorithms for distributed snapshots and global virtual time approximation. *Parallel Disrribured Computing*, 18, 1993.

[65] M. Ahuja, A.D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 12 –19, may-1 jun 1990. doi: 10.1109/ICDCS.1990.89327.

Appendix A

CPSML Grammar

⟨cpsmodel⟩→ ⟨model⟩ ⟨colon⟩ ⟨name⟩ ⟨decl⟩ ⟨cys⟩ ⟨phys⟩ [⟨ae_behavior⟩]

⟨name⟩→ ⟨id⟩

⟨decl⟩→ ⟨pe_decl⟩ ⟨area_decl⟩

⟨pe_decl⟩→

'AE' 'Declaration' 'Begin'

⟨ae_decl⟩

'AE' 'Declaration' 'End'

'RS' 'Declaration' 'Begin'

⟨rs_decl⟩

'RS' 'Declaration' 'Begin'

⟨ae_decl⟩→

⟨id⟩ **'('** ⟨prop_list⟩ **')'** ⟨colon⟩ ⟨instance_decl⟩

| ⟨id⟩ **'('** ⟨prop_list⟩ **')'** ⟨colon⟩ ⟨instance_decl⟩ ⟨ae_decl⟩

⟨rs_decl⟩→

⟨id⟩ **'('** ⟨prop_list⟩ **')'** ⟨colon⟩ ⟨instance_decl⟩

| ⟨id⟩ **'('** ⟨prop_list⟩ **')'** ⟨colon⟩ ⟨instance_decl⟩ ⟨rs_decl⟩

⟨instance_decl⟩→

⟨id⟩ **‘(’** ⟨prop_list⟩ **‘)’** ⟨semi_colon⟩

| ⟨id⟩ **‘(’** ⟨prop_list⟩ **‘)’** ⟨comma⟩ ⟨instance_decl⟩

⟨prop_list⟩→ ⟨id⟩ | ⟨id⟩ ⟨comma⟩ ⟨prop_list⟩

⟨area_decl⟩→

‘Area’ ‘Declaration’ ‘Begin’

⟨area_decl1⟩

‘Area’ ‘Declaration’ ‘End’

|

⟨area_decl1⟩→

⟨id⟩ ⟨colon⟩ ⟨area_instance_decl⟩

| ⟨id⟩ ⟨colon⟩ ⟨area_instance_decl⟩ ⟨area_decl1⟩

⟨area_instance_decl⟩→

⟨id⟩ ⟨semi_colon⟩

| ⟨id⟩ ⟨comma⟩ ⟨area_instance_decl⟩

⟨cys⟩→

‘CyS’ ‘Begin’

⟨ce⟩ ⟨e⟩

‘CyS’ ‘End’

⟨ce⟩→ **‘CE’** ⟨equal⟩ **‘(’** ⟨ce_list⟩ **‘)’** ⟨semi_colon⟩

⟨ce_list⟩→

⟨ceid⟩ **‘(’** ⟨p_list⟩ **‘)’**

| ⟨ceid⟩ **‘(’** ⟨p_list⟩ **‘)’** ⟨comma⟩ ⟨ce_list⟩

⟨p_list⟩→

⟨pid⟩

| ⟨pid⟩ ⟨comma⟩ ⟨p_list⟩

⟨e⟩→ **‘E’** ⟨equal⟩ **‘(’** ⟨e_list⟩ **‘)’** ⟨semi_colon⟩

⟨e_list⟩→ ⟨eid⟩

| ⟨eid⟩ ⟨e_list⟩

⟨phys⟩→

‘PhyS’ ‘Begin’

(⟨flat⟩ | ⟨pat⟩ ⟨re⟩ **)**

‘PhyS’ ‘End’

⟨flat⟩→ ⟨gp⟩

⟨gp⟩→ ⟨pa⟩ ⟨re⟩

⟨pa⟩→ **‘PA’** ⟨equal⟩ **‘(’** ⟨pa_list⟩ **‘)’** ⟨semi_colon⟩

⟨pa_list⟩→

⟨phaid⟩

| ⟨phaid⟩ ⟨comma⟩ ⟨pa_list⟩

⟨re⟩→ **‘RE’** ⟨equal⟩ **‘(’** ⟨re_list⟩ **‘)’** ⟨semi_colon⟩

⟨re_list⟩→

⟨reid⟩

⟨reid⟩ ⟨comma⟩ ⟨re_list⟩

⟨pat⟩→

⟨id⟩ **‘contains’** ⟨children⟩

| ⟨id⟩ **‘contains’** ⟨children⟩ ⟨pat⟩

⟨children⟩→

⟨id⟩ ⟨semi_colon⟩

| ⟨id⟩ ⟨comma⟩ ⟨children⟩

⟨ae_behavior⟩→

‘AE’ ‘Behavior’ ‘Begin’”

⟨ae_behavior_list⟩

‘AE’ ‘Behavior’ ‘End’”

⟨ae_behavior_list⟩→

⟨id⟩ ⟨colon⟩ **‘(’** ⟨ae_list⟩ **‘)’** ⟨semi_colon⟩ ⟨statements⟩ | ⟨id⟩ ⟨colon⟩ **‘(’**

⟨ae_list⟩ **‘)’** ⟨semi_colon⟩ ⟨statements⟩ ⟨ae_behavior_list⟩

⟨ae_list⟩→

⟨id⟩

| ⟨id⟩ ⟨comma⟩ ⟨ae_list⟩

⟨interaction⟩→

⟨variable_return⟩

| ⟨non_return⟩

| ⟨list_return⟩

⟨variable_return⟩→

‘acquire’ **‘(’** ⟨id⟩ **‘)’**

| **‘receive’** **‘(’‘)’**

⟨non_return⟩→

‘move’ **‘(’** ⟨id⟩ **‘)’**

| **‘release’** **‘(’** ⟨id⟩ **‘)’**

| **‘send’** **‘(’** ⟨message⟩ **‘)’**

| **‘sense’** **‘(’** ⟨id⟩ **‘)’**

| **‘lose’** **‘(’** ⟨id⟩ **‘)’**

| **‘change_state’** **‘(’** ⟨id⟩ **‘)’**

⟨list_return⟩→ **‘observe’** **‘(’‘)’**

⟨message⟩→

‘{’

‘ID’ ⟨equal⟩ ⟨int⟩ ⟨comma⟩

‘payload’ ⟨equal⟩ **(** ⟨character⟩ | ⟨digit⟩ **)***

‘}’

⟨list_expr⟩→

⟨list⟩ ⟨dot⟩ **‘add’** **‘(’** ⟨id⟩ **‘)’**

| ⟨id⟩ ⟨equal⟩ ⟨list⟩ ⟨dot⟩ **‘remove’** **‘(’** ⟨id⟩ **‘)’**

| ⟨list⟩ ⟨equal⟩ ⟨list⟩ ⟨dot⟩ **‘ae’**

| ⟨list⟩ ⟨equal⟩ ⟨list⟩ ⟨dot⟩ **‘rs’**

| ⟨list⟩ ⟨equal⟩ ⟨list⟩ ⟨dot⟩ ⟨id⟩

⟨list⟩→ ⟨id⟩ **‘[]’**

⟨assign⟩→

⟨id⟩ ⟨equal⟩ ⟨expr⟩

| ⟨id⟩ **‘++’**

| ⟨id⟩ **‘–’**

⟨expr⟩→ ⟨mult_expr⟩ **((‘+’** | **‘-’)** ⟨mult_expr⟩**)***

⟨mult_expr⟩→ ⟨atom⟩ **(‘*’**⟨atom⟩**)***

⟨atom⟩→

⟨int⟩

| ⟨id⟩

| **‘(’** ⟨expr⟩ **‘)’**

⟨comparision_op⟩→ **‘<’** | **‘>’** | **‘<=’** | **‘>=’** | **‘==’** | **‘! =’**

⟨logical_and_or⟩→ **‘and’** | **‘or’**

⟨logical_not⟩→ **‘not’**

⟨size_of_op⟩→ **‘sizeof’ ‘(’** ⟨list⟩ **‘)’** ⟨comparision_op⟩ ⟨expr⟩

⟨conditional_expr⟩→

⟨size_of_op⟩

| **(** ⟨expr⟩ ⟨logical_and_or⟩ ⟨expr⟩ **)** **(**⟨logical_and_or⟩ ⟨expr⟩**)***

| **(** ⟨id⟩ ⟨dot⟩ ⟨dot⟩ **‘state’ ‘==’ (‘free’** | **‘busy’))**

| **(** ⟨id⟩ **‘==’ (‘true’** | **‘false’))**

| ⟨logical_not⟩ **‘(’** ⟨conditional_expr⟩ **‘)’**

⟨statements⟩→

⟨statement⟩*

|

⟨statement⟩→

⟨assign⟩ ⟨semi_colon⟩

| **‘break’** ⟨semi_colon⟩

| **‘continue’** ⟨semi_colon⟩

| **‘goto’** ⟨id⟩ ⟨semi_colon⟩

| ⟨id⟩ ⟨colon⟩ ⟨statement⟩

| ⟨non_return⟩ ⟨semi_colon⟩

| ⟨list⟩ ⟨equal⟩ ⟨list_return⟩ ⟨semi_colon⟩

| ⟨id⟩ ⟨equal⟩ ⟨variable_return⟩ ⟨semi_colon⟩

| ⟨id⟩ ⟨equal⟩ ⟨message⟩ ⟨semi_colon⟩

| ⟨list_expr⟩ ⟨semi_colon⟩

| ⟨while_loop⟩

| ⟨if_stmt⟩

| ⟨select_edge⟩ ⟨semi_colon⟩

| ⟨for_each⟩

| ⟨id⟩ ⟨dot⟩ **‘state’** ⟨equal⟩ **(‘free’** | **‘busy’))** ⟨semi_colon⟩

⟨while_loop⟩→

‘while’ ‘(’ conditional_expr **‘)’ ‘{’**

⟨statements⟩

‘}’

⟨if_stmt⟩→

‘if’ ‘(’ conditional_expr **‘)’ ‘{’**

⟨statements⟩

‘}’ ((‘else’ ‘{’

⟨statements⟩

‘}’) | **(‘else’** ⟨if_stmt⟩**))?**

⟨select_edge⟩→ ⟨id⟩ ⟨equal⟩ **‘select’ ‘unvisited’ ‘edge’ ‘from’ ‘RE’**

⟨for_each⟩→

‘for’ ‘(’ ‘each’ ⟨id⟩ **‘in’** ⟨list⟩ **‘) ‘{’**

⟨statements⟩

‘}’

⟨model⟩→ **‘Model’**

⟨colon⟩→ **‘:’**

⟨dot⟩→ **‘.’**

⟨semi_colon⟩→ **‘;’**

⟨comma⟩→ **‘,’**

⟨equal⟩→ **‘=’**

⟨ceid⟩→ **ce**⟨int⟩

⟨eid⟩→ **e**⟨int⟩_⟨int⟩

⟨pid⟩→ **p**⟨int⟩

⟨phaid⟩→ **a**⟨int⟩

⟨reid⟩→ **re**⟨int⟩_⟨int⟩

⟨digit⟩→**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

⟨id⟩→ **(‘a’..‘z’|‘A’..‘Z’|‘_’) (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’)***

⟨type⟩→ **(‘a’..‘z’|‘A’..‘Z’|’_’) (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’)***

⟨int⟩→ **(‘0’..‘9’)+**

⟨newline⟩→ **‘\r’?‘\n’**

⟨ws⟩→ **(‘ ’|‘\t’|‘\r’|‘\n’|‘\r’‘\n’)+**

⟨line_comment⟩→ **‘//’ (‘\n’|‘\r’)*‘\r’?‘\n’**

Appendix B

Sample CPSML Programs

```
Model: Flat_Model
  AE Declaration Begin
    User(name, ID) : U1(user1, A1), U2(user2, A2), U3(user3, A3), U4(user4,
    A4);
  AE Declaration End

  RS Declaration Begin
    Resource(ID): R(RS1);
  RS Declaration End

  CyS Begin
    CE = (ce1(p1), ce2(p2), ce3(p3), ce4(p4));
    E = (e1_2, e2_3, e1_4, e2_4, e3_4) ; //this is comment
  CyS End

  PhyS Begin
    PA = (a1, a2, a3, a4);
    RE = (re1_4, re4_2, re2_3, re4_3);
```

```
PhyS End
```

Listing B.1: *Sample CPSML program modeling the system shown in Figure 1.1(b)*

```
Model: PAT_Model
  AE Declaration Begin
    Nurse(name, ID): N1(Wendy, N12), N2(Rowdy, N23);
    Doctor(name, ID): D1(John, D22);
    Patient(name, ID): P1(Jay, P223), P2(Shawn, P554), P3(Stuart, P5676);
  AE Declaration End

  RS Declaration Begin
    WheelChair(ID) : WC1(WC1), WC2(WC2);
    XRay(ID): X1(X1), X2(X2);
  RS Declaration End

  Area Declaration Begin
    Hospital: Hospital1;
    ICU: ICU1;
    Floor: Floor1;
    NurseStation: NS1;
    RestRoom: RR1, RR2, RR3, RR4;
    Room: Room1, Room2;
    WaitingArea: WA1;
    Fine: A1, A2, A3, A4, A5, A6,
        A7, A8, A9, A10, A11, A12,
        A13, A14, A15, A16;
  Area Declaration End

  CyS Begin
    CE = (ce1(p1), ce2(p2), ce3(p3), ce4(p4), ce5(p5), ce6(p6),
      ce7(p7), ce8(p8), ce9(p9), ce10(p10), ce11(p11), ce12(p12),
```

```
    ce13(p13), ce14(p14), ce15(p15), ce16(p16));
  E = (e1_2, e1_4, e1_9, e1_10, e4_9, e4_8, e4_6, e4_2,
    e4_3, e4_5, e4_11, e4_12, e12_11, e12_5, e5_3, e5_13,
    e3_15, e13_14, e14_16, e15_16, e15_5) ;
CyS End

PhyS Begin
  Hospital1 contains Floor1 ;
  Floor1 contains ICU1, RR1, RR2, Room1, Room2, NS1, WA1, A8, A9, A10, A4,
 A5 ;
  ICU1 contains A15, A16;
  Room1 contains A12, RR3;
  RR3 contains A11;
  Room2 contains A13, RR4;
  RR4 contains A14;
  RR1 contains A7;
  RR2 contains A6;
  ICU1 contains A15, A16;
  NS1 contains A2, A3;
  WA1 contains A1;
  RE = (re1_2);
PhyS End

AE Behavior Begin
  Nurse: (N1, N2);
  L1: pe[] = observe();
    resources[] = pe[].rs;
    if(sizeof(resources[]) != 0) {
      for (each r in resourcess[]) {
        if(r.state == free) {
          goto L2;
```

```
        }
60      }
    }
62  n = select unvisited edge from RE;
    move(n);
64  goto L1;
L2: val = acquire(r);
66  if(val == false){
      goto L1;
68  }

70  Doctor: (D1, D2);
L1: pe[] = observe();
72  resources[] = pe[].rs;
    if(sizeof(resources[]) != 0) {
74    for (each r in resourcess[]) {
        if(r.state == free) {
76        goto L2;
        }
78    }
    }
80  n = select unvisited edge from RE;
    move(n);
82  goto L1;
L2: val = acquire(r);
84  if(val == false){
      goto L1;
86  }
AE Behavior End
```

Listing B.2: *Sample CPSML program modeling the system shown in Figure 1.2*

Index