A PROGRAMMING MODEL FOR SAFER PERVASIVE SPACES

By

CHAO CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012

To my Mom

ACKNOWLEDGMENTS

I would like to express the deepest gratitude to my committee chair, Professor Sumi Helal, who has been patiently advising me and persistently helping me in the past several years. Without his guidance, caring, and patience, this dissertation would not have been possible.

I also would like to thank my committee members, Professor Shigang Chen, Professor Paul Fishwick, Professor My T. Thai, and Professor Raymond Issa, for their guidance and constructive comments during the course of this dissertation research.

In addition, I give my thanks to all colleagues in the Mobile and Pervasive Computing lab. Special thanks to Duckki Lee, who as a good friend, is always willing to help and offer his best suggestions.

Finally, I would like to thank my families. My parents, Su Liu and Zhigang Chen, and my wife, Hui Zhou, were always standing by me and giving unconditional support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application programming interface
DDL	Device Description Language
DIOS	Domain-independent Ontology for Safety
DOST	Domain ontology for Sensor Taxonomy
GTSH	Gator Tech Smart House
HCI	Human computer interface
IOT	Internet of Things
OWL	Web ontology language
RDF	Resource description framework
RFID	Radio frequency identification
SOA	Service-oriented architecture
XML	Extensible markup language

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

A PROGRAMMING MODEL FOR SAFER PERVASIVE SPACES
By

Chao Chen

December 2012

Chair: Abdelsalam (Sumi) Helal
Major: Computer Engineering

In a world where pervasive computing technology permeates every corner of our lives, safety is paramount. Unlike traditional computer systems, pervasive systems tie intimately with the users. Errors and conflicts in such space could have detrimental, dangerous or undesired effects on the space, the devices, and more importantly, the user. There are no support systems or programming models conscious of the issue of safety. Unrestrained programming is the model de jour, which is inadequate. A programming model is needed to encourage and obligate various roles engaged in the development of pervasive spaces to contribute to increasing safety.

This research presents a programming model that utilizes role-specific safety knowledge, and takes advantage of the rich sensing and actuation capabilities of pervasive systems to prevent or detect/avert “impermissible contexts”. A domain-independent ontology for safety (DiOS) is presented, which provides a uniform semantic basis for representation of safety knowledge and concerns. We follow by introducing DDL, a practice to augment a device description language with DiOS vocabulary. We show how DDL/DiOS enables a path to integrate domain-specific safety knowledge using a universal semantic framework.

Our proposed programming model also attempts to bring the well-established concept of transaction to the realm of pervasive computing. We propose pervasive transaction, a novel concept that serves as an abstraction of the cyber-physical interactions in pervasive space. We further identify a set of properties: atomicity, integrity, isolation and durability (known as AI^2D-safety properties), which together define the safety criteria for runtime pervasive transaction behavior. The concepts of pervasive transaction and AI^2D safety reveal a path to improve the overall safety of pervasive spaces through transaction management - safety risks can be prevented, or detected/mitigated by regulating the runtime behavior of pervasive transactions. As an implementation of such runtime mechanism, we present the context lock protocol, which introduces a number of relaxations to suit the practical need of pervasive transactions. We prove the correctness of the protocol by showing it always produces transaction histories that satisfy the AI^2D-safety criteria. We conclude this work with an analytical model that evaluates the performance of the context lock protocol. We present the model derivation and numerical results.

CHAPTER 1
PROBLEM STATEMENT

Introduction

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

—Mark Weiser, *The Computer for the 21st Century*

The essence of that vision was the creation of environments saturated with computing and communication capability, yet gracefully integrated with human users.

—M. Satyanarayanan, *Pervasive Computing: Vision and Challenges*

The vision of pervasive computing, as described by Mark Weiser [22] in 1991, and elaborated by M. Satyanarayanan [23] in 2001, promised great opportunities to our future as individuals, societies and organizations. But these opportunities do not come without higher risks inherent in the nature of the highly instrumented spaces of deployment. The extensive interactions between technology and human, raises concerns over a number of issues, especially on safety and human well-being [3]. Without a thorough understanding of the safety implications of pervasive computing technology, it is too risky for this community to move beyond prototypes and trials in controlled environments and rush into large-scale practical system deployments with actual impacts, based on blind optimism. Two decades after Mark Weiser's vision was unveiled, there remains a need to revisit and reflect on what this vision entails.

Fusion Of The Cyber And Physical Worlds

The extensive interactions between the pervasive system and the physical world, enabled by the massive deployment and use of sensors and actuators, makes it possible for programmers to create pervasive applications that deliver convenient

services and critical assistances in many aspects of our lives. However, the capability of a pervasive system to interact and influence the physical world also implies that the impact of malfunctioned systems or ill-programmed applications is no longer limited to loss of data, waste of time and effort or a crash of a computer; the risk of various harmful ramifications to the users and damages to the physical properties are as real as those services the pervasive system intends to provide.

Invisible Technology

Pervasive computing technology is expected to fade into the background of our lives to become calm and even invisible [4]. While invisibility is a much-desired merit when pervasive services are quietly and smoothly performing their daily duties, it creates a sense of lost-control when a pervasive system disconnects itself from users under the circumstances of failures, hazards and other critical conditions. Without a proper channel of human-computer interaction, the calmness of computing, in turn, may result in hectic reactions from its human users.

User-centric Computing

Many pervasive systems are intimate [5][6], because they have extensive interactions with users, and perform personal tasks on behalf of them. Intelligent environments such as smart homes and intelligent vehicles keep extensive records of their users and act upon their preferences and limitations. These systems exist in homes and cars, which are considered to be private, safe and treated with affections. Many services involve our daily routines, including medicine reminder, scheduler, or even personal hygiene. They are integrated into users' daily lives, receiving high expectations and trust from users. The intimacy between human and pervasive systems

magnifies the sense of frustration when these services fail to perform or even cause damages [3].

The pervasive computing community has been spending great efforts to achieve remarkable progress on research issues such as device integration, service discovery, security and privacy. However, the issue of safety in pervasive spaces largely remains an uncharted area [7][10]. Most of the residential spaces and work sites nowadays adopt some form of safety codes and standards [8][9]. In spite of the extensive deployment of pervasive infrastructures and devices, most of these spaces are still relying heavily on human consciousness and constant effort for compliance and enforcement of these safety protocols. Sensor technologies introduce great opportunities for monitoring and detecting critical conditions but have been put in limited use. Typical applications such as smoke detectors and burglary alarms are often packaged as closed systems by their vendors, and are therefore not programmable and hardly configurable to satisfy varying safety needs. While there is an imminent need in today's technology market for a programming model that promotes open and flexible compositions of sensors and actuators to provide extended and customizable safety services, existing programming models suffer from lack of checks and balances on programmers, which themselves may open greater loopholes that invite safety hazards and damages to the space [10].

Unleashing the Programming Power of Pervasive Computing

As new sensing and actuation technology continue to emerge, possibilities for new pervasive computing applications are endless. Even conventional devices and appliances are being re-purposed to serve completely non-intended use. For example, compact fluorescent lights have been recently shown to be useful as sensors to detect

human presence [69]; and cellphone microphones have been creatively used as health sensors to detect lung functions [72]. An infrastructure-mediated sensing approach is therefore emerging which proposes to reuse existing infrastructures, such as HVAC systems [71], and conventional gas, waster or electricity meters to sense events and human activities [70].

However, realizations of these ideas rely on one important pre-requisite - the pervasive systems need to be programmable. Devices, services, and infrastructures in general, should provide external accessibility through open interfaces (APIs), to support and encourage composition and collaboration with others, and to enable novel and more sophisticated services. A programmable pervasive system, as the above examples promised, will no doubt improve utilization, foster better and more innovative applications, achieve enhanced user experience, improved quality of life, and increased return on investment.

In reality though, we have been building spaces that were intentionally designed with limited programmability. Spaces have been instrumented with separate systems, each working in isolation, to accomplish a fixed set of functions. Systems such as HVAC, gas and electricity meters, fire alarms, and security monitors are regarded as "stovepipes" - closed, hardly evolving, and inaccessible. An important reason for such stovepipes is the concern of safety - as systems becomes more open, new uncertainties will be introduced, which either creates new types of safety hazards, or impairs the existing critical services that these systems were designed for.

The goal of this research, therefore, is to unleash the potential of pervasive computing by safely “opening” these "stovepipes" - we aim to create a programming

model for pervasive systems that encourages interactions and collaborations among services, and at the same time addresses the safety concerns by enabling safety enforcement and assurance.

In the remainder of this chapter, we first take a closer look at the potential safety problems by presenting a few scenarios. Then we survey the existing programming models and discuss why they are risk-prone or inadequate to address safety problems. We conclude this chapter by defining the scope of this dissertation.

Safety Issues in Pervasive Spaces

A typical pervasive space is a physical space instrumented by a number of sensors and actuators, which are managed by a middleware and its software components, to provide a variety of services and assistances to users [37]. In addition to the common safety hazards [35][36] in mundane environments, pervasive spaces are often exposed to greater risks because of ill-programmed software services or unreliable hardware components of pervasive systems [34]. In this section, we present a few scenarios to exemplify the issue of safety in pervasive spaces.

Scenario A - Over-frequent Actuation

The smart entrance service is an abstraction of a door opener and provides a *doorOpen()* method. Everything looks solid until an excited programmer calls this method once every 5 seconds, only to find that the thin metal latch, which the service is truly representing, cannot withstand the repeated fatigue and finally breaks.

Scenario B – Hardware Failure

The smart entrance service consists of a proximity sensor to detect visitor's arrival, a floor sensor to confirm visitor's entrance, and an actuator to open / close the door. When a visitor arrives, the service opens the door. However, the floor sensor runs out of

battery, and could not confirm the visitor's entrance. As a result, the execution of the service is suspended in the middle, creating an unsecure situation by leaving the door open.

Scenario C – Service Conflict

The smart entrance service is modified to accommodate people with disabilities. It holds the door open for an extended period of 30 seconds before it closes. The energy conservation service, unaware of this change, forces the door to close after opening for 10 seconds, causing potential injury to the user.

Scenario D – Changing Requirements

The programmer of the smart entrance service could hardly call it a day, even after he fixes the problems in Scenario A, B and C – apparently, everyone has something to say about the door. The owner of house wants the door to remain closed as much as possible to save on the energy bill; the resident asks the door to be automatically secured during sleeping time; even the newly bought in-door grill, whose safety instructions state "door should remain open to avoid fume and smoke". At the end of day, the programmer has to sew up these fragmented and potentially conflicting pieces of knowledge in his software, and investigates on the grill to figure out how both services could work together. What is worse, he has to be ready for new challenges as any new device, user, service joining the house could result in a major modification to his software. What seems to be a quick and easy job turns into a laborious and daunting mission that never ends.

The above scenarios are only a few examples of the numerous potential safety risks created by pervasive spaces. While programs are responsible to ensure the correct and safe behaviors of their applications, it is not fair to lay the entire burden on

the shoulders of programmers. As shown in the above scenarios, even when each individual service and application is correctly executed, risk conditions may still rise as a result of device failure, service conflict, or human and environmental factors. In addition, the lack of safety support in existing programming models invites unrestrained programming practices and often results in awkward and inadequate safety solutions. A new programming model is needed to enhance the programmer's consciousness of safety and encourage more stakeholders to engaging in the creation of safer pervasive systems.

Programming Models for Pervasive Spaces

By providing simple but powerful abstractions, programming models support a broad category of programmers to develop applications, services and a variety of administrative tools and system monitors [33]. These abstractions hide the complexity of pervasive systems away from programmers, enabling easier and more convenient programming practices. However, these programming models could also blind programmers with improperly designed abstractions that limit their understanding of the safety details of pervasive spaces. On one hand, some programming models provide excessive flexibility to programmers, encouraging a constrained style of programming, which may invite safety loopholes. On the other hand, the over simplification of other programming models may raise the concerns of losing the expressive power when it comes to specify detailed safety knowledge in their programs. In this section, we survey the existing programming models for pervasive and discuss their inadequacies in safety.

The service-oriented programming model, a popular paradigm for developing pervasive computing systems and applications, encourages a free and unrestricted style of application composition over available services [6][7]. While this is a powerful

capability to the programmer, it may lead to unpredictable behaviors in the pervasive space when the applications are deployed. In theory, a programmer can select any set of services from the service pool and connect them to form an application. Service methods can therefore be invoked in any order. Clearly, such lack of constraints on service composition may unintentionally but adversely cause conflicts and contentions among the various services. This could create a potentially enormous state space of the system and opens up possibilities for states that are undesirable, unpredictable or even impermissible.

An extended version of service-oriented programming model has been proposed by [3]. To protect against false and unsafe states of a service, the model defines a prioritized safety API, which mandates the service programmer to supply safety-checking instructions and specify preemptive emergence handling methods through a uniform service interface. While this approach improves the safety conditions of individual services and devices, it is less effective to detect and resolve competing or conflicting scenarios, which involve multiple services. In conclusion, this model focuses on safety conformance of each individual service, but overlooks the risks brought by the intricate relationships among services such as dependence and conflicts.

The context-driven programming model, takes an alternative approach to address the safety issue by monitoring unsafe conditions at a global level [38][39]. Undesired and unsafe situations are specified as impermissible context in a global context grid that describes the overall status of the space. At run time, the stream of sensor readings are examined to verify if any safety violation has taken place. While maintaining and monitoring a global context grid could be expensive, a greater problem is, how to create

such context grid and align it with the various safety concerns in the first place. Furthermore, the context-driven programming model encourages programmers to delegate all responsibilities of safety checking to the centralized context grid, while many safety concerns could be better internalized and more effectively addressed within the service or the application.

The event-driven model is similar to the context-driven model, but provides a "Context-Condition-Service action"–like rule syntax for programmers [2][40][41]. The syntax requires a rule-processing engine in the pervasive system to constantly monitor for events and conditions, which may incur a significant performance overhead.

Figure 1-1 demonstrates the tradeoff between the power and expressiveness of a programming model and the safety margin it provides. Context driven model is shown to provide high safety index but falls short in its ability to program many desired applications. On the other hand, service oriented models offer very high programmability but they fall short in being unrestrained and inadequate to ensuring safety. The figure also shows where the community should invest its efforts – in the envelope of both increased programmability and safety.

Scope

This research aims at enhancing the safety of pervasive spaces with a new programming model focused on safety. In our model, the notion of "programming" has been extended. In addition to programmers, more stakeholders are obligated to engage in the design, development and management of pervasive spaces. In particular, we achieve the enhanced safety by accomplishing the following:

- Devise a new design and development practice that can enhance the safety awareness of not only programmers, but also other providers and suppliers involved in the creation of pervasive spaces.

- Development of a new theory that models the execution of pervasive systems from a safety-oriented perspective, and defines a set of practicable properties that ensure the safe execution of pervasive systems.

- Design and implementation of a run-time mechanism based on the above-mentioned theory, that can effectively prevent, or detect and mitigate safety risks in pervasive spaces.

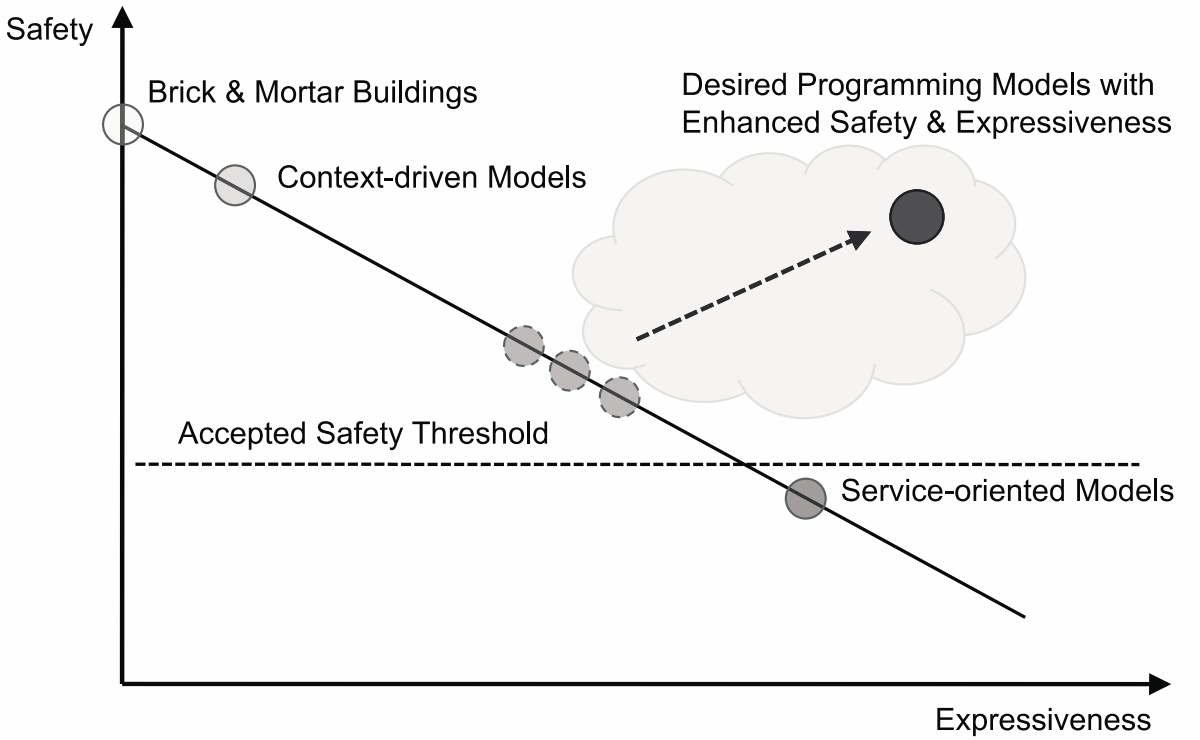


Figure 1-1. Tradeoff between safety and expressiveness of programming models in pervasive spaces.

CHAPTER 2
LITERATURE REVIEW

Introduction

Safety, along with other issues such as security and privacy, has been placed into the research agenda of the pervasive computing community for almost a decade. While we have seen a plethora of literature over security and privacy research, the outcome of safety research remains limited. In a survey of pervasive computing systems conducted in 2010 [37], we found only a few research projects that include safety as their direct objective [27][28][38]. The concept of safety in pervasive computing remains loosely defined, and is sometimes used interchangeably with terms such as dependability and trustworthiness, which are compound system properties that involve reliability, availability, security, and safety [41][42][43][44]. In fact, most of these research works rely heavily on reliability/availability-based approaches, which stem from the conventional safety engineering. A brief review of safety engineering in conventional industries (in the first section) will show why these concepts/approaches are inadequate in addressing safety concerns in pervasive environments. Then we present of a brief survey (in the second section) over a number of safety-related research projects for pervasive computing systems.

This chapter reviews the existing literature in two directions. First, we survey the recent research in the conventional safety-engineering domain. A comparison study is conducted to discuss the common safety approaches among several conventional industries. Through comparing the safety issues in conventional industries environments with pervasive spaces, we show why conventional safety engineering

approaches is inadequate. Second, we survey the published work on safety and safety-related issues that focus on the domain of pervasive computing.

Safety Engineering in Conventional Industries

Safety is not a novel issue created by the recent pervasive computing technology. In fact, the concept of safety has been closely tied with engineering ever since the discipline of engineering starts to take shape [56]. As technology continues to evolve, safety remains a primary concern and goal for today's engineering products, whether they are brick and steel structures, electrical machineries, or sensor-based pervasive systems. The domain of safety engineering has long been established to extensively investigate the issue of safety in a wide range of industries such as transportation, chemical processing, nuclear power, and construction, etc. To prevent, detect, mitigate or recover from the various safety risk scenarios, a plethora of techniques and devices have been invented, codes and regulations have been established, many of which are essential to ensure the safety of today's pervasive spaces.

However, pervasive technology also brings forth new challenges for safety. The extensive interactions between cyber and physical worlds extend software failures and logic errors to real damages and physical harms. Furthermore, the close connections between human and computing, further magnify the risks to users when the safety of pervasive computing is compromised. These challenges calls for new perspectives and approaches to safety, which can better utilize the sensor/actuator-rich feature of pervasive computing.

The issue of safety has been studied extensively for many decades in various conventional industries such as commercial aviation, rail transportation, chemical processing, etc. In this section, we briefly survey the conventional approaches to safety

in each of the above industries. By comparing their various approaches, we attempt to (1) separate the concept of safety from reliability, (2) distinguish the pervasive space engineering from conventional industries, and (3) lay out the new safety challenges in pervasive spaces.

Commercial Aviation

The safety of commercial aircraft largely depends on reliability [46], which is mainly achieved by (a) single element integrity – aircraft components are designed and manufactured to meet the FAA-approved failure rate requirement, and (b) redundancy – aircraft are equipped with redundant key components to cope with single or multiple point failure. Aircrafts typically have long product cycles, allowing new models to be carefully improved based on previously known causes of accidents. Aviation industry in general is very conservative towards new technology and always avoids drastic changes in their product design.

Rail Transportation

Rail transportation is one of the safest forms of on-land travel [57]. Unlike aircrafts, the majority of rail accidents are caused by collisions. Therefore the most important safety measure is train-signaling system, which controls train traffic in the railway network to create separation among trains [53]. Modern train-signaling systems use computers to calculate safe zones, or “blocks”, around each moving train so that no other train is allowed to enter. The safety and correctness of train control software can be verified using model checking tools [58].

Chemical Processing

The chemical processing industry mainly focuses on three types of safety hazards: fire, explosion, and toxic release [46]. The maturation of design and operation

procedures over the years has led to industrial standards and codes to eliminate or contain these hazards. In chemical industry, most safety approaches emphasis on component reliability and add-on protection devices. However, with the use of new technology, and the increase in scale and complexity of controlling systems, the focus of safety has changed from individual components reliability to the overall system integrity [47]. The growing risks of human errors and false system interactions have lead to system accidents reported in [59].

Safety vs. Reliability

As shown in the above survey, reliability has been regarded as an integral property contributing to the safety of overall systems in most industries. Reliability engineering, which concerns primarily with failures, has become a dominant safety approach in conventional industries. Reliability engineer uses a variety of techniques to minimize component failures and thereby the failures of overall systems caused by component failure, including parallel redundancy, standby sparing, built-in safety factors and margins, screening, and timed replacements [46].

Although engineers tend to use the terms of reliability and safety alternatively in many cases, safety and reliability are in fact two different concepts. Increased reliability do not necessarily guarantees increased safety, since not all safety hazards are created by failures. In fact, safety is a broader concept than reliability, as failures may or may not compromise safety. As the size and complexity of a system increases, the risk of human errors, unsafe interactions, and conflicts can grow significantly, jeopardizing the safety of the overall system even if individual component failures are kept at a low rate.

Pervasive Space Engineering vs. Product Line Engineering

Unlike the matured conventional industries that typically have product-lines providing end-to-end solutions to manufacture and assemble final products, pervasive space industries are yet to deliver a complete package of smart home technologies as a product. The idea of “smart home in a box” has not come into realization. The creation of a pervasive space so far is largely an ad-hoc process that requires lots of hacking and tuning. The lack of standardization in the development process poses additional challenges to safety engineering for pervasive spaces. Unlike conventional products that have fixed life cycles and a finalized design at the early stage [49], pervasive systems often continue to evolve even after its deployment. With new services and devices continues to enter the pervasive space, the purpose and goals of the space also start to shift, leading to a changing requirement of safety. Meanwhile, new risks and conflicts could rise as a result of the changed configuration. Therefore safety can no longer be addressed in a single step of the product design, but needs to be constantly managed during its entire lifecycle. As an example, Table 2-1 provides a summary of the comparisons between an aircraft cockpit and a smart home.

Research on Safety and Safety-related Issues in Pervasive Computing

Research on Safety-related Issues

There are two main areas of research related to safety in pervasive computing: system fault-tolerance and programming safety.

Fault-tolerant systems address reliability in pervasive spaces by concentrating on detecting, mitigating and containing “faults” – failures of components in pervasive systems. The GAIA Project [60] at the University of Illinois at Urbana Champaign

proposed a fault tolerant mechanism that detects device failures in pervasive spaces using “heartbeats” – beacon signals periodically sent by devices to announce their availability. Fault mitigation and containment is typically achieved by redundancy. The Gator-Tech Smart House Project [61] at the University of Florida proposed the “virtual sensor” concept, which is a logical representation of a group of physical sensors with similar functions. Virtual sensors provide fault resilience by substituting faulty sensors with others in the same group. Another branch of fault-tolerant systems is “self-healing systems”, which aim to provide automatic recover process transparent to end-users [62]. The PACE Project [63] at University of Queensland suggested a fully automatic reconfiguration and replacement method for fault recovery, owing to their unique execution model. An application of their system reaches context information along a path called process chain, which is a pipelining of execution units called processes. Such a process chain is starting from physical sensors called atomic processes, and constructed dynamically at runtime, by an extraction from a predefined process relationship graph. Hence, if a sensor has a fault but no other sensor in the same category is available; the process chain is reconstructed dynamically based on other category of sensors.

The programming safety approaches focus on programming languages and tools to enhance software robustness and safety of pervasive systems. Chen and Motet proposed a technique to embed system safety requirement as control structures in the programming language [47]. Kelly and Weaver introduced the Goal Structure Notation (GSN) [61] as a formal representation for safety cases in complex systems [50]. Martins et al. investigated in type safety in programming language for wireless sensor networks

[48]. Programming structures utilizing exception-handling mechanism have also been brought to pervasive systems to handle failures and unsafe situations [43][64][65].

Safety Research in Pervasive Computing

Despite of a plethora of literature on safety-related issues such as reliability, dependability, security, privacy and trust, not many researchers have directly addressed the issue of safety of pervasive systems. Among the papers claimed safety solutions to pervasive spaces, many narrowed the scope of safety issue and only provided solutions from a particular aspect of pervasive computing. For example, de Champs et al. focused on the application layer where individual pervasive software and device could be verified using model checking techniques [55]. Baskar investigated the interactions among devices and argued conflicts and interferences among devices constituted major risk factors for pervasive environment. He proposed using “safety controllers” as a set of decentralized device regulators to enforce the safety policies in pervasive environment [54].

Ma et al. first introduced the concept of “Ubisafe Computing”, a computing paradigm aiming to construct a conflict-free and safe pervasive computing environment using a unified methodology [68]. To define the scope of safety, a few more concepts were coined such as u-object (including u-system and u-person) and u-environment. A u-person is considered to be non-negative if he/she does not attack other u-systems or u-persons. Safety is defined by the absence of negative relationships among u-objects. However, in this vision paper, no solutions were proposed to address the technical challenges in the paper, such as how risks and attack can be detected.

Similarly, Yang and Helal identified the four fundamental elements of safety in pervasive space: device, user, service and space [3]. A service-oriented programming

model was proposed where the safety requirement of the other three elements (device, user and space) are embedded in their corresponding service representations [21]. A safety-enhancement mechanism has been devised to mandate all services to implement a strict safety interface and register their risk handlers.

Table 2-1 Comparison between aircraft cockpit and smart home.

	Aircraft Cockpit	Smart Home
System Architecture	Closed	Open
Safety Goals	Fixed	Changing over time
Components	Tightly coupled; High reliability; FAA regulated	Loosely coupled; Variable reliability; Non-standardization or regulated
Business Model	Well-established models	Roles and models are yet to shape up
Evolvability	Finalized products; Fully tested	Frequent updates, upgrades, & customization; Continue to evolve after deployment

CHAPTER 3
PROBLEM ANALYSIS

Safety of pervasive spaces is a broad concept, which involves a multitude of elements ranging from human users to physical environment, from software services to hardware devices. The overall safety of a pervasive space requires not only satisfaction of the various safety requirements for individual elements, but also protection over the extensive interactions among these elements.

Safety is also a constant concern spanning all stages during the lifecycles of a pervasive system, which requires a collaborative effort from a variety of roles in the design, development, and management of pervasive spaces.

In this chapter, we analyze the breadth and depth of the safety problem in pervasive spaces, by presenting the results of two preliminary studies. We first dissect the pervasive spaces and summarize four fundamental elements of safety. We then introduce a collaborative model that defines the safety responsibility of multiple roles involved in the lifetime of a pervasive space. These studies help us define the scope and challenges of the problem, and prepare us for an overall safety approach, which will be presented at the end of the chapter.

Fundamental Elements of Pervasive Spaces

To understand the scope of safety problem in pervasive spaces, first we need to understand what a pervasive space entails, and where the safety concerns arise from. A pervasive space is a physical space instrumented with sensor and actuator devices, which are managed by software applications and services, to provide assistances to human users. Yang and Helal [3] first propose the four fundamental elements of

pervasive spaces: device, service, user and space. In this section, we analyze each of the four elements and discuss their safety implications.

Device

The capabilities to interpret and interact with the physical world start with devices. They are the I/O devices that bridge the gap between the physical world and digital realm. Although the complexity, capability, size and other attributes may vary, a device can always be classified as either a sensor or an actuator. Sensor acts as the source of data, which reports the status of the surrounding environment or the device itself back to the system. Actuator, on the other hand, functions as the recipient of commands, with which the system-issued instructions are translated into certain action sequences and carried out accordingly. More complicated devices such as smart appliances or robots, which are capable of sensing data as well as receiving commands, can be considered as a combination of sensors and actuators.

The notion of safety for a device is two-fold. First, the device itself is at a sound state and secure from damage, which requires the device to be deployed in a device-friendly environment and operated with proper and safe procedures. Second, the device does not jeopardize the safety of the overall space (known as a pervasive space), which includes other devices and services, users and the physical environment. This requires an individual device to be conscious of the global safety constraints when it interacts with the physical space.

Service

Service is at the core of any pervasive computing system. The term service is overloaded with various meanings that a clarification is required before we can commence on the following discussion. With any non-trivial computing system, we

expect the premeditated favorable outcome or useful functionality to be delivered upon the activation of software artifacts or systems. These functionalities and outcomes can be delivered by a stand-alone software artifact or in coordination with others as a group. We use the term service to describe the collection of software artifacts that are involved in the process as well as the sequence of events in completion of this deliverance.

Each service is usually designed to fulfill very specific purposes, with very specific requirements on the availability of other services or the existence of configurations in order to accomplish its goals. In the dynamic environment of pervasive computing, a service presents one of the smallest and simplest entities that can be well-regulated.

User

As in any system modeling and analysis, attempts to model human behavior is laborious but yet rarely generate any accurate results. User, as one of the fundamental elements of pervasive computing system, cannot be as easily and clearly modeled as devices and services. But users are critical because they have the final say in deciding what the system need to do, and sometimes even how they should be done. Such is true even in those systems aiming at complete automation.

Users are the most delicate element in the pervasive space in terms of the tolerance of any risk factors. Hence a model without user will be incomplete and utterly useless. The bottom line is the systems exist to serve users, and there is no reason for any system to exist if there was nothing users want to achieve.

Space

Most researchers do not consider the space itself as a critical element of pervasive computing system, but rather only as a collection of services and devices, as well as the interactions among themselves and between them and users. This assumption often

leads the programmers to see the trees but miss the forest. A space is an important element because the status of the space as a whole often provides critical contexts for decision making; a space also encompasses all other elements within, in particular the status of services, devices and users as well as the interactions between them; side affects usually cannot be captured by devices or services, but can be described by measurable changes in a space. Based on functions, social contexts and the locale of a space, there are different restrictions and interpretations of the contexts. For instance, the interpretation of cleanness in a clean room will be different from a butcher shop; the acceptable temperature and humidity also drastically differ in an engine room from an operation room.

As shown in Fig. 3-1, the four elements enabled a new perspective on the problem of safety in pervasive spaces. Applying the “divide-and-conquer” strategy, we find that the overall safety condition of a space can be improved by protecting its devices from misuse and failures, restraining services from risky and hazardous behaviors, satisfying user’s preferences and safety needs, and maintaining additional safety constraints imposed by the space itself. To break down of the overarching safety concerns into element-wise measures and knowledge, we need to first identify the roles for the various stakeholders, such as device manufacturers, service programmers, users and domain experts, who possess a broad range of expertise on each of these elements.

Roles and Collaboration Models for the Design, Creation, and Management of Pervasive Spaces

Building a pervasive space involves many skilled people. In our lab, it took a team of technicians, programmers, electronic engineers, computer scientists and domain experts working closely with the owners of the space, to design and develop the Gator

Tech Smart House [15]. While it is exciting and enjoyable to have interdisciplinary collaborations in a university environment, few pervasive system developers and space owners would have such luxury to bring together all these skill sets and expertise in the commercial environment.

In practice, pervasive technology enjoys a healthy yet fragmented industry market, spanning sensor manufacturers, connected medical and personal devices, the suppliers of networks, middleware and applications for sensor connectivity and integration, and many other smaller pieces of the puzzle.

In industry, many companies will concentrate on specific elements of the entire pervasive system. Numerous device manufacturers such as Analog Devices [16] and A&D [17] produce hardware ranging from pin-head sensors and actuators to more complex devices and smart appliances. A number of middleware providers such as Pervasa [18] and Augusta Systems [19] create middleware platforms that connect applications with devices; and there are all manner of software development enterprises writing applications for a variety of systems.

Others take more systematic approaches by providing end-to-end solutions that focus on specific solutions in the pervasive space. For example, ActiGraph is a company specialized in human activity monitoring. Its products include not only wearable sensors for activity measurement, but also software tools for data collection, aggregation and analysis. Some companies extend their offerings further by providing human services like monitoring centers, coupled with software and hardware in their solution. For instance, most burglary alarm businesses provide installation and monitoring services.

This is a large market filled with many complementary players who bring a wide range of strengths and expertise. What is needed is a playbook, guidelines allowing individual suppliers to position themselves and team with others. While it may look obvious that different roles must coordinate to develop a pervasive system, it is not clear how responsibilities and resources should be divided, or how coordination and collaboration is achieved among these roles. Without an accurate articulation of these roles, an architecture that supports the separation of roles and, a careful design of a business model that inter-relates them, it will be increasingly difficult to integrate each party's participation. In this article, we propose a development model of pervasive spaces consisting of several roles that undertake different responsibilities at various stages in the life cycle of a smart space. We see the roles in such a development model supported by well-designed, loosely coupled system architectures, based on standards that enable the creation and interoperability of these spaces.

Early mainframe computers such as the IBM S/360 and ICL 2900 knew only two roles: programmers and operators. As the computer evolved to networked servers and workstations, users and system administrators' roles were added. And today, pervasive systems seem to require new and revised roles to enable a development model that works despite a fragmented technology market and that can utilize existing IT experience in supporting the pervasive space throughout its entire life cycle.

The need of a new collaboration model is even more paramount when we consider the problem of safety, which is broad issue spanning the lifetime of the space from design to operation stage. In addition to common physical hazards such as fire and burglary, a pervasive space has many more monitoring concerns: failure of devices,

contention between services, and the safety and well being of users in the space, just to name a few. The distributed nature of safety hazards and protocols is another driver for a correlation of events and a collaboration model that ties in multiple roles. For example, the specification of safety constraints requires combined knowledge from the space owners and domain experts. Compliance with safety rules, in turn, sets obligations on device manufactures, application developers and users. The detection and elimination of safety risks requires collaboration among sensing devices, monitoring personnel, actuation equipment and certain middleware mechanisms. We need a model that provides a clear separation of responsibilities among these roles and offers a path toward seamless collaboration on development, deployment and the complete lifecycle of pervasive spaces. In the following sub-section section, we introduce an ecosystem in which multiple roles can work together to design, develop and maintain pervasive systems with improved safety.

A role is a socially expected behavior pattern. It embodies a set of connected rights and obligations. To create a collaboration model for building pervasive spaces, we need first to specify the resources and responsibilities of each role in the model. The resources include internal assets such as knowledge, professional skills and budgets, as well as external ones such as device specifications, programming APIs and software toolkits provided by other roles or third parties. The responsibilities describe what is expected from each role to contribute to the pervasive space. Figure 3-2 gives an overview of these roles and their relationship in a collaboration model. Some of the identified roles have already been well established in the industry, while others are

terms coined for positions we consider critical to building a pervasive smart space. In this section, we introduce each role in more detail and provide a summary in Table 3-1.

General Space Developer

General space developers are the general contractors, experts in overall pervasive space development. Qualified general space developers possess the knowledge and experience necessary to take charge of design, implementation and deployment of the pervasive system. They understand the client's needs and have the resources to acquire and integrate various components, and to implement solutions which address those needs. Typically, the space developer works closely with the space owner to formulate requirements that meet the owner's goals. This will drive supplier decisions such as which middleware would best serve as a framework for the system. They are responsible for purchasing services and devices that satisfy user's need. To connect and integrate these components, the space developer hires technicians for the physical installation and deployment. Once the smart space is up and running, the developer hands over the system to the space administrator for testing and final acceptance. The general space developer is a critical role that coordinates other roles at the various stages of development. Due to the diverse nature of pervasive spaces in different application domains, general space developers can be expected to specialize in different types of spaces.

Space Admin

Our experience has convinced us that the essential role of maintaining upgrading and reconfiguring the smart space is often overlooked in its design and development [20]. A pervasive system does not stop evolving after its initial deployment. While it is unrealistic to retain the entire development team to make adjustments, a space

administrator will have to cope with frequent device changes and service upgrades. This role covers typical monitoring and response services, and extends to the handling of emergencies and other events in the smart space, which require human intervention or instruction. The space administrators' role is to monitor and administer a pervasive space. They evaluate, approve and deploy newly introduced services, and provide first response to safety alerts and other emergency situations. Also they coordinate with space technicians, service providers and device manufactures for service upgrades and everyday maintenance.

Space Technician

A space technician is a sub-contractor working under the general space developer. The technician role covers many tasks in the construction of the cyber/physical space infrastructure, such as setting up networks, installing servers, controllers and personal computers, and, wiring the sensors and actuators. They follow the space's specifications designed by the general developer and provide the bulk of the physical deployment and integration work.

Device Manufacture

A device manufacturer supplies products such as sensors, actuators, controllers and networks that constitute the physical layer of a pervasive system. To make a device available to the various services and applications, its manufacturer is obligated to provide detailed specifications of device interfaces and protocols to other roles such as service providers. These specifications can be written in a standard descriptor language such as DDL. The middleware provider or a third party then provides software tools which convert specifications into software services, allowing access and control to the device via standards-based service interfaces.

Middleware Provider

Middleware serves as a running environment to host and integrate a variety of software components. To achieve seamless integration of both software services and hardware devices, the middleware provider is responsible for supplying standards-based service APIs, a programming IDE to service developers, as well as tools and mechanisms to device manufacturers to assist them in fulfilling the responsibility of device descriptions (as explained in the section 2). In addition, middleware has to provide tools and user interfaces allowing a space administrator to install, configure, and maintain the system at runtime.

Space Service Provider

The role of space service providers is to provide implementations of software services based on goals and requirements as specified by the space owner and general space developer. This could entail implementation efforts for developing new or highly customized services, or the porting and reuse of pre-existing services. Sometimes, a service depends on the presence or the absence of other services to work properly. Therefore, its providers carry the responsibility to explicitly state this dependence relationship so that the general space developers and administrators will make informed decisions to include or exclude certain services in various configurations.

As pervasive systems permeate the globe, it is imperative to develop collaborative models that enable the various suppliers and providers to efficiently contribute and integrate the resources needed to create smart and manageable spaces. In addition, the concept of “separation of concerns” is needed so that each participating role is assigned clear-cut responsibilities. Our collaborative model is an attempt to separate

the safety concerns by assigning safety obligations to a variety of roles at different stages of the lifetime of a pervasive system.

Table 3-1. Distribution of resources/responsibilities among roles

Roles	Business Roles	Resources	Responsibilities
Space Owner	Customer	Budget	Specify goals Approve and accept Hire general space developer and admin
General Space Developer	General Contractor	Expertise in space and system development	Requirement analysis Architectural design Sub-contract with other roles
Space Technician	Sub-contractor	Skilled labor and services	Physical deployment & integration Everyday maintenances
Space Administrator	Provides Services to Customer	All inputs, outputs, and configuration options available from Space	Runtime monitoring Administers space configuration & software updates; First response to critical events
Middleware Provider	Vendor	Expertise in pervasive middleware	Provide facilities for service integration
Device Manufacturer	Vendor	Knowledge of its product	Specify devices
Space Service Provider	Software Service Component Provider	Programming API	Specify dependencies on devices and other services

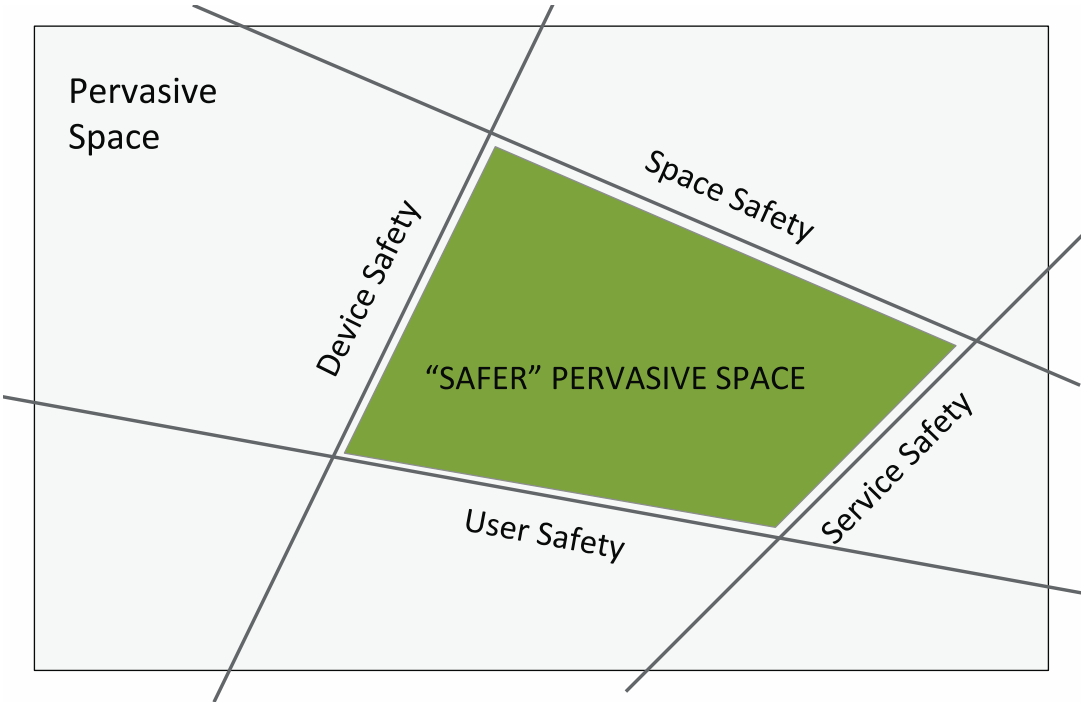


Figure 3-1. Define a safe pervasive space element-wise. The set of safe states of a pervasive space is the intersection of the safe states of each element in this space including device, service, user and the space itself.

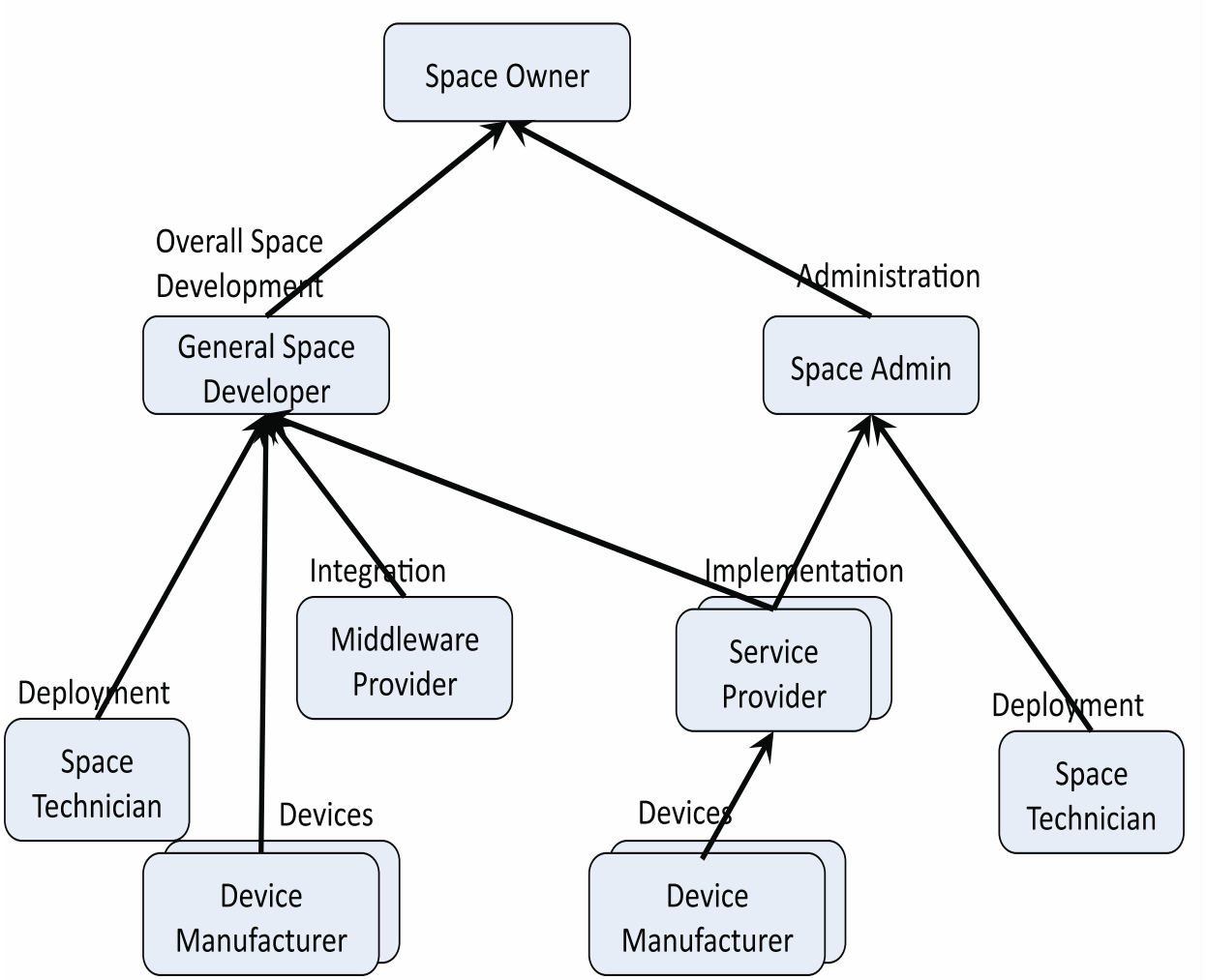


Figure 3-2. Overview of the multiple-role collaboration model in pervasive space development and management.

CHAPTER 4
OVERALL APPROACH

Why do We Need an Overall Approach?

Safety problems are extremely diverse in pervasive spaces, so are the techniques utilized to detect, prevent or recover them. Can we protect the overall safety of a pervasive space, by simply combining these techniques to address each and every safety issue we can identify? In this section, we argue that such ad-hoc solutions will not be feasible, and that an overall systematic safety approach is necessary, to address the diverse safety needs in an open and complex pervasive environment.

An ad-hoc approach typically addresses a specific safety issue with a closed set of risk scenarios. This approach works well with many closed systems that have fixed use cases, however, it is not flexible enough to address the safety need in an open pervasive environment whose applications may be constantly evolving. In addition, ad-hoc approaches often utilize dedicated devices and services, which are not meant to be shared with other applications. The lack of programmability on these closed boxes further impedes their ability to adapt to new and customized safety requirements.

Ad-hoc safety approaches also do not scale well with pervasive spaces. As the numbers of users, devices, and services in pervasive spaces increase, the number of cyber-physical interactions grows exponentially, which opens a vast amount of possibilities for new risk scenarios. Addressing each safety risk by a new ad-hoc safety technique seems inefficient and wasteful. It is more desirable to have an overall approach where devices, services, and knowledge can be reused to handle new safety issues. In addition, conflict and contention becomes a major safety risk factor as a consequence of the rich interactions in pervasive spaces. Ad-hoc approaches, often

intended for closed sub-systems run in isolation, cannot address this category of safety risks. Therefore, a systematic approach to manage, regulate, and protect the concurrent interactions between the physical space and cyber world is needed.

Challenges

As we quest for an overall approach to address the safety issue, we encounter a number of challenges.

First, integration of safety knowledge and concerns. A systematic safety approach requires a uniform understanding of safety based on the collective knowledge from different stakeholders from various domains. The lack of a standard safety language impedes the integration, consolidation and utilization of these safety knowledge and concerns.

Second, enforcement of safety rules and constraints. Unrestricted interactions between pervasive systems and physical spaces could create undesirable consequences violating the safety needs of the space. A runtime enforcement mechanism is needed to reduce the risk of these violations.

Third, evaluation of safety mechanisms. Enforcement of safety constraints can be achieved in different ways at various costs. Finding the most cost-effective way requires performance evaluations on safety enforcement mechanisms. Identifying performance metrics, synthesizing benchmarks, and establishing performance models could be challenging.

The Approach

This dissertation presents a safety-oriented programming model that aims to address the above challenges. Our programming model embodies two main ideas – (1) a domain-independent ontology for safety, which addresses the fragmentation of safety

knowledge and concerns by proposing a universal vocabulary for safety concepts, that empowers various stakeholders, in addition to programmers, to formulate safety constraints in uniformed semantics. This allows the creation of an integrated safety knowledge base, which provides safety guidelines for runtime safety mechanisms. (2) a novel concept of pervasive transaction, which models the interactions between the cyber and the physical space as transactions. This concept provides possibility to utilize database transaction processing theories to manage the safety of pervasive spaces.

A Domain-independent Ontology for Safety

The main idea of a domain-independent ontology for safety is to enable different parties to collaboratively specify and seamlessly integrate their safety concerns over various aspects of the pervasive space. As we discussed in Chapter 3, pervasive computing systems can be dissected into four fundamental elements, each identified as a source of risk that carries certain safety concerns. Based on these elements, we create an ecosystem to establish several roles through the lifecycle of a pervasive space. These roles, separated by distinct responsibilities and resources, collaboratively contribute their own safety concerns and knowledge by using a small and scalable ontology over the universal concerns of safety. This ontology is independent of any application domain and offers a unified expression of safety constraints. We name this ontology as DiOS (domain-independent ontology for safety). With DiOS, a stakeholder in the ecosystem is able to specify safety constraints in universal safety concepts. DiOS can also be used as an extension to a domain specific language, such as device descriptor or user profiling languages. This will enable specification and integration of safety requirements from parties of various application domains.

Pervasive Transaction

The idea of pervasive transaction originates from an interesting analogy that we observed between pervasive systems and conventional database management systems. A database controls the use and maintains the integrity of the database, which is a collection of shared data objects. Database transactions perform read and write operations to access and manipulate these data objects concurrently. Similarly, a pervasive system is deployed in a certain physical space, whose state is observable and controllable through sensing and actuations, which are logically equivalent to read and write operations. This analogy reveals a leverage opportunity to approach the safety issue in pervasive spaces in a way similar to approaching atomicity, consistency, integrity, and durability (ACID) in database.

Analogous to database transactions, we propose the concept of pervasive transactions as a general programming model for safe application development in cyber-physical spaces. In this model, we use context to represent the state of the physical world. A pervasive transaction therefore is simply a sequence of read/write operations over context, which results in a context transition of the pervasive space. Since according to our approach, pervasive transaction is the only form of interaction between the cyber world and the physical world, we can improve the overall safety of the pervasive system by runtime transaction management. Safe transactions will be processed and committed. Risky transactions will be restrained and/or aborted. And conflicting transactions will be isolated. We took the idea of pervasive transaction one-step further, by proposing context lock - a runtime mechanism that protects the physical space by “locking” the context within its desired range. Context lock achieves isolation among pervasive transaction through access control. By adhering to a locking protocol,

execution of conflicting transactions will be separated in time and resolved. We further employ context locks in enforcing useful notions of durability in pervasive spaces. Finally, we propose safety transaction - a system-controlled transaction issued by pervasive systems to recover the space from a risky context, and/or safely abort risky transactions.

The Scope

This dissertation research aims at providing system-wide safety support to pervasive spaces (shown in Figure 4-1). To address the above challenges, we propose a two-fold approach. First, we present an ontology-based programming model, which enables the fusion of scattered and fragmented safety concerns and knowledge from a broad range of roles into an integrated safety knowledge base. Second, to enforce safety constraints from the integrated safety knowledge, we devise a runtime mechanism to regulate the operation of pervasive systems based on what we call the theory of pervasive transaction.

The core of the programming model is a small but scalable ontology over the universal concerns of safety. This ontology allows roles from different domains to express distinct knowledge and concerns using the same safety vocabulary. With this ontology as a semantic base, we are able to embed unified expressions of safety constraints in device specifications, user profiles, service descriptions, and space configurations, and collect them in an integrated safety knowledge base.

The run-time enforcement mechanism bases itself on the theory of pervasive transaction, which establishes an abstract model of the interactions between pervasive systems and physical spaces. In this theory, the execution of a pervasive application is modeled as a transaction, consisting of a number of sensing and actuation operations.

Our run-time mechanism ensures the maximal conformance of safety constraints by regulating the execution of pervasive transactions. In the event of safety hazard, the run-time mechanism also composes and invokes safety transactions to restore the pervasive space to a safe and stable state.

The rest of this dissertation is organized as following. We introduce DiOS, a domain-independent ontology for safety in Chapter 5. Then we present the Device Description Language (DDL) in Chapter 6, as a case study of utilizing DiOS ontology to collect safety concerns and knowledge from devices. In Chapter 7, we introduce a formalization of safety using the concept of pervasive transactions. The detailed implementation of pervasive transactions is presented in Chapter 8. In Chapter 9, we evaluate the performance of our implementation by presenting an analytical model and numerical results. We conclude this work with a summary in Chapter 10.

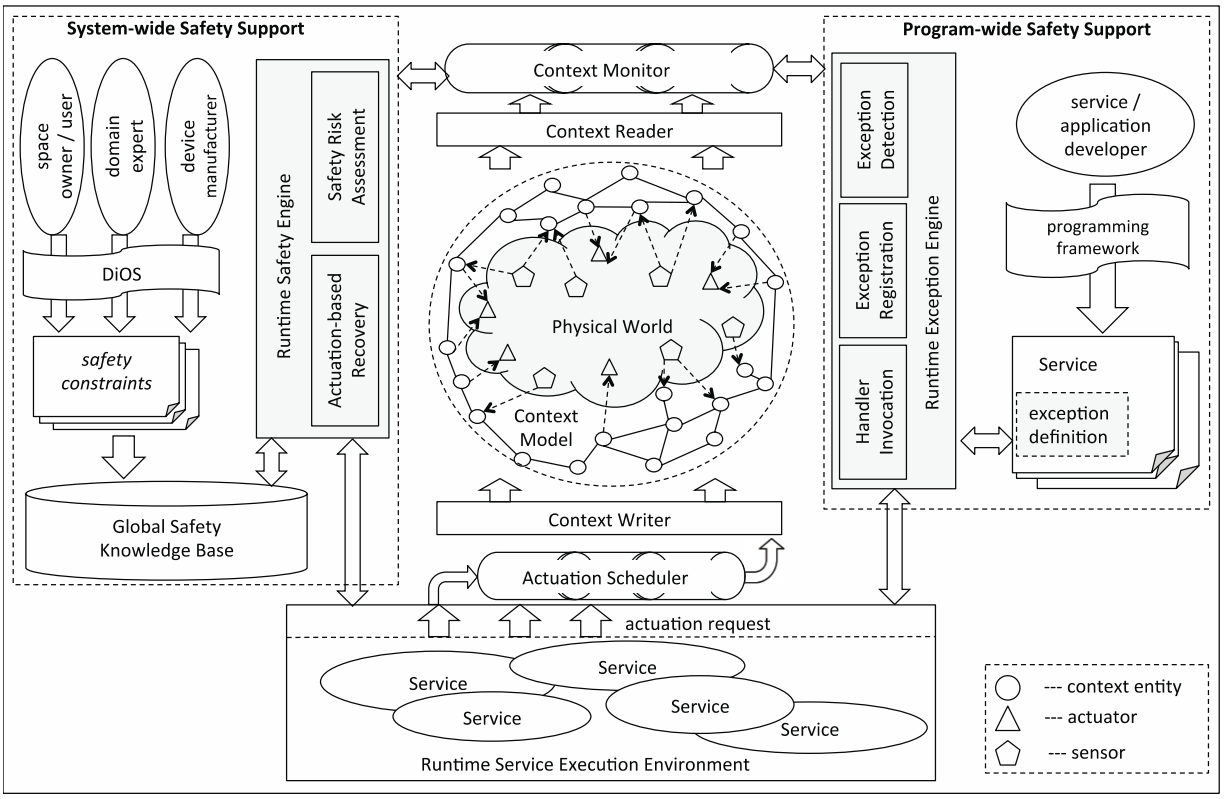


Figure 4-1. Overview of safety architecture for pervasive spaces, which is divided into system-wide (left) and program-wide (right) safety support.

CHAPTER 5
DIOS: A DOMAIN-INDEPENDENT ONTOLOGY FOR SAFETY

Creating pervasive spaces requires us speaking a new language of safety. Given the scale and complexity of most pervasive systems, there cannot be a single designated role for safety. A clearly defined common safety vocabulary is needed to allow a variety of programmers, providers and suppliers to exchange, share and eventually integrate their safety knowledge and concerns.

The Domain-independent Ontology for Safety (DiOS) is small but scalable ontology that defines the universal safety concepts independent of the domains of applications. DiOS consists of two distinct yet related ontologies: DiOS Core and DiOS Periphery. DiOS Core defines the core vocabularies for specifying safety concerns and knowledge. DiOS Periphery, extended from the core ontology, establishes a generic model of pervasive spaces, which includes various elements pertinent to the issue of safety. Figure 5-1 presents the overview of the DiOS ontology.

DiOS Core

The root of the DiOS Core ontology is the OWL class *ctx: Context*. Context is a conceptual abstraction of the physical state of pervasive spaces. The simplest form of context is a single context entity with a value. For example, “temperature (context entity) at 75 degrees (value)” is a piece of context. Context can be classified as safe or unsafe based on the domain knowledge of the experts. The rest of the DiOS Core ontology is divided into two parts: (a) a *safety classification* of context, which defines the semantic terms to separate context according to the level of safety, and (b) a taxonomy of *context compositions*, which defines a set of key concepts constituting a context. We describe each part of the ontology in the following subsections.

Safety Classification

To describe the safety level of a particular context, DiOS defines four safety classifiers: *risky* for context with potential damage, *severe* for elevated risk and ongoing damage, *desirable* for safe and preferred context, and *neutral* for context that is neither unsafe nor desirable. Both risky and severe context are *impermissible* context, whereas desirable and neutral context are considered as *permissible* context. Therefore, the overall safety of a pervasive space can be described as the conditions of all context entities. A space is considered unsafe if an impermissible context exists.

Safety is a complex issue involving both subjective perceptions and objective facts. On one hand, personal feelings and desires affect one's assessment of safety. A piece of context carries different safety implications and entails different actions, when it is assessed by people with different perspective, domain knowledge, and interest. On the other hand, the risky (or safe) phenomena represented by the context are factual and independent of personal opinions. In addition, the pervasive system needs a coherent view of the overall space safety despite of the separated concerns and interest of the various stakeholders (such as users, space owner, device manufacturers, and service providers) in the space. Therefore it is important to make sure our ontology accommodates both the subject and objective nature of safety. When a stakeholder uses the classification system to address a safety concern, there should be a clear separation between the physical reality and the personal opinion.

Another problem involving the design of the safety classification is to justify its adequacy. We frequently face questions such as: How detailed the safety classification system should be in order to be adequate? Why not add (or remove) a safety level from DiOS? Would a quantitative approach be more precise than quantitative classification?

To answer these questions, we based our design on the following considerations. First, safety is a *spectrum*. From the very safe and desirable, to extremely risky and hazardous, we can always divide safety into many varying degrees depending on various criterions. No matter how many levels we have, we could always design a classification that is more detailed and precise. Even if we have a classification system that is perfectly adequate for one application, we can never guarantee it is equally adequate for another. Therefore, we should shift our focus from adequacy to *utility*. Our goal here is to design a minimal but universal separation that is useful and reasonable for all applications. We come up with four safety categories in our design based on the discrepancy of both urgency and system reaction. With the urgency of situation changes, the pervasive system changes its mindset and responds in different ways. This serves as a guideline for stakeholders to address their concerns at proper safety levels, establishing reasonable expectations of the system behavior should a safety risk occur.

- *Desirable Context*. Desirable context are states not only safe but also preferred by its stakeholders such as users and the space owner. Under a desirable context, a pervasive system is required to do nothing but trying to maintain this context. A piece of desirable context and transitions within desirable contexts will not trigger any drastic or urgent responses from the system.

- Neutral Context. Neutral context denotes states that are under protection but not preferred by any stakeholders. Neutral context are considered to be safe. While a transition from a neutral context to a desirable context is preferred, it is not required. Therefore, like desirable context, a neutral context will not trigger any drastic or urgent responses from the pervasive system. However, the system may upgrade a neutral context into a desirable context.

- Risky Context. Risky Context describes states when safety risks occur and create pending damage or harm to the space. A pervasive space under risky context is no longer considered to be safe although the actual damage has not occurred. Risky context presents the best opportunity for a pervasive space to recover and avoid lost of property or life. The pervasive system is required to respond

immediately to all risky contexts. The actions taken to convert a risky context to a permissible context are called lossless recovery.

- Severe Context. When risk elevates and causes actual damage, the hazardous states constitute Severe Context. Like risky context, severe context is also impressible and must be responded immediately. Under a severe context, a pervasive system can either retreat to risky context or a recover to a permissible context.

Context Composition

In DiOS, context is an overarching concept that describes the state of the entire pervasive space or a subset of the space. A context can be described by the state of a single context entity, a context entity set, or a situation. Each of these three concepts is defined by its corresponding class in the DiOS ontology document.

- *Context Entity*. The OWL class *ctx: Context Entity* defines a logical entity that represents elements of various natures in the pervasive space, including conceptual entities such as software services, and physical entities such as objects, devices, users, and the physical environment. To describe the status of a *Context Entity*, each entity has several attributes (*Entity Attribute*) taking values in their corresponding domains. For example, the physical environment is a *Context Entity* with *Entity Attributes* such as temperature, humidity, and luminance. Therefore, we can characterize the state of the physical environment by a vector of its attribute values, e.g. [temperature, humidity, luminance] = [76 F, 50%, 50 cd/m^2]. In a pervasive space, the values of these *Entity Attributes* are acquired by sensors and other approaches such as software probes and logs.

- *Context Entity Group*. A context in pervasive spaces can also consists of a group of context entities. The OWL class *ctx: Context Entity Group* defines a set of related context entities. Context entity groups are usually used to characterize certain phenomenon or events, which involve a group of context entities, each changing its own states to reflect such phenomenon or events. For example, DUI (driving under influence) can be characterized by the combination of the driver's physiological signs such as blood alcoholic level and the car's state such as speed and orientation. In addition, within a context entity group, there are attributes describing the relationship among entities. These attributes do not belong to any individual entity, and are defined as group attributes. For example, the distance between two objects is a group attribute when the objects are in the same context entity group.

- *Situation* [43]. Situation is a temporal sequence of context. The *ctx: Situation* class has a member class *ctx*:*Context Entity Group* for the group of context it is keeping track of, and a property class *ctx:Situation Attribute* that describes time information

of the context, such as the timestamp of each context snapshot, and the time interval in between context.

DiOS Periphery

DiOS Periphery presents a context-based abstraction of pervasive systems. A pervasive system contains two types of devices: sensor and actuator. A sensor performs sensing operations, which reads from its corresponding context entity, while an actuator performs actuations, which modifies the corresponding context entities. Any pervasive application, therefore, is abstracted as a series of read and write operations over various context entities.

DiOS Periphery defines a generic model of the pervasive space and provides a top-level taxonomy of the elements in the space. It also defines the relationship between these elements and the DiOS Core ontology. DiOS Periphery is domain-independent yet domain-aware. The model of pervasive spaces provided by this ontology is independent of any applications, making minimum assumptions on the system architecture and implementation of the space. (For example, DiOS models the unit of software components in pervasive systems as “services”, alluding to a Service-Oriented Architecture.) At the same time, DiOS Periphery is aware of the various application domains and offers an open framework for further extensions from domain-specific ontologies. In DiOS Periphery, domain-specific concepts will be subsumed and aligned with the generic ontology, making it possible to represent domain-specific concerns and knowledge within a domain-independent framework.

The root of the DiOS Periphery ontology is the OWL class *pvs: Pervasive Space*, which is also the root of the taxonomy of the pervasive space elements. The *Pervasive*

Space class contains three top-level elements: (i) *Physical Space*, (ii) *User*, and (iii) *Service*.

Physical Space

The *pvs: Physical Space* class defines vocabularies for describing the physical aspect of the pervasive space such as its temperature, humidity and luminance. Typically a pervasive space has exactly one instance of the physical space, which can be described by a set of its environment attributes. To align this ontology with DiOS Core, the *Physical Space* class is defined as a *Context Entity*, while the *Environment Attribute* is defined as an *Entity Attribute*. Depending on the interest and requirement of individual spaces, the set of environment attributes of a space is extensible to include customized, domain-specific attributes. For example, in a bathroom environment, it is interesting to know the temperature of both the air and the tub water. Hence, we can add *airTemperature* and *waterTemperature* as two sub-classes extended from the attribute *Temperature*.

User

User is an important element in addressing the safety issues in pervasive spaces. On one hand, the profile of a user provides valuable information that may reveal the user's preference on safety settings. Information such as age, gender, medical histories, and social identities can be utilized to infer desired or unwanted contexts of a user. On the other hand, context information such as user's motion, gesture, facial expression, and physiological conditions themselves are often critical signs of risky and unsafe situations. The *usr: User* class consists of typical vocabularies for describing both the static profile and the dynamic status of a person. To align with the DiOS Core ontology, class *User* is defined as a subclass of *Context Entity.*

Service

The third element, service, is a key concept to model the system (cyber) side of the pervasive space, which sets itself apart from the physical space. We define service as a unit of system component that delivers a set of functionalities by invoking its service methods. An important property of class *Service* is *Service Method*. We define a service method by the transition of context caused by invoking the method. The context before a method invocation is defined as *Precondition*, and the context after its invocation is defined as *Effect*.

Although our service ontology is alluding to a service-oriented architecture (SOA) employed in the pervasive space, it does not necessarily mandate one. In fact, the concept service is loosely defined in DiOS and intended to cover similar concepts such as process and task in other programming models and architectures.

Device Service

Although services are typically considered as software entities, a hardware device can also become a service when it is properly wrapped by a software representation. We define this type of service in class *Device Service* as a subtype of *Service*.

DiOS Periphery is both domain-independent and domain-aware. The ontology establishes a model of the pervasive space that is independent of any domain of applications, making minimum assumptions on the architecture and implementation of the pervasive system. Meanwhile, DiOS Periphery offers an open framework for further extensions from specific domain. The top-level taxonomy in DiOS Periphery, defines a set of generic concepts, allowing domain-specific knowledge such as user profiles and device specifications to extend from with the generic ontology. Therefore, DiOS make it

possible to integrate domain-specific concerns and knowledge within a domain-independent framework.

Summary

DiOS provides the semantic basis for uniform specifications of safety rules and constraints. In the following chapter, we show how we extend a device description language with DiOS vocabularies to specify safety concerns and knowledge for devices.

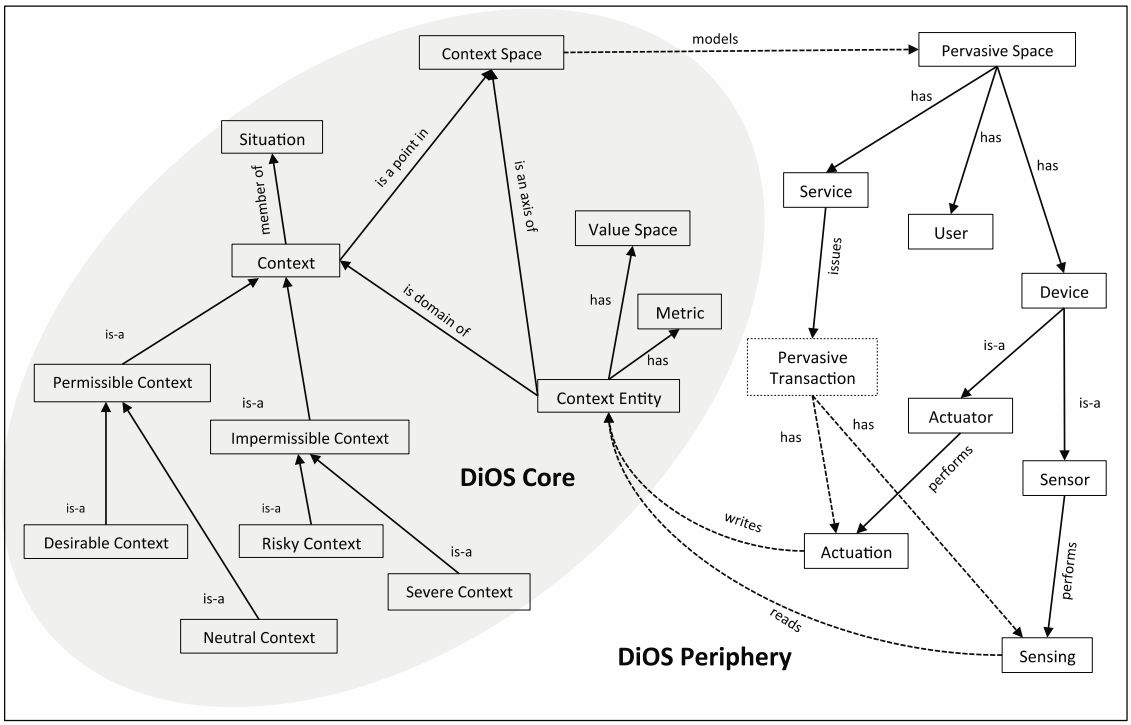


Figure 5-1. Overview of the DiOS ontology.

CHAPTER 6
DDL: UTILIZING DEVICE INFORMATION FOR SAFER PERVASIVE SPACES

Introduction

With the massive deployment of sensors, actuators, and everyday objects enhanced with communication and computing capabilities, pervasive computing is pervading almost every aspect of our daily lives, raising ever-growing concerns for safety. As pervasive systems close the gap between cyber world and physical spaces by weaving technology into “the fabrics of everyday life” [22], they also create an imminent risk of extending logical failures and computing errors to physical damages and real harms, because of their capability to interact and influence the physical world. Restraining this capability from turning into harmful even destructive forces requires our attention on its technology enablers – sensor and actuator devices. This chapter aims to address the issue of safety in pervasive spaces from a device-centric perspective.

From a device’s point of view, the notion of its safety is two-fold. First, the device itself is at a sound state and secure from damage, which requires the device to be deployed in a device-friendly environment and operated with proper and safe procedures. Second, the device does not jeopardize the safety of the overall pervasive space, which includes other devices and services, users and the physical environment. This requires an individual device to be conscious of the global safety constraints when it interacts with the physical space. By protecting the safety of each individual device in operation, we expect to enhance the overall safety of the entire pervasive system.

An interesting observation is made when we use the concept of context to represent the state of pervasive spaces. The safety of pervasive spaces relies on the ability of pervasive systems to (1) monitor and discover risky context through sensors,

and (2) actively rectify or mitigate unsafe context through actuators. Sensors and actuators together, while presenting challenges to the safety of pervasive space as points of concerns, also reveal the opportunity as part of the solutions to this very issue. The goal of this chapter, therefore, is not only to restrain sensors and actuators from risky and unsafe operations, but also to unleash the power of the same devices by engaging them in the monitoring and protection of the pervasive spaces.

In this chapter, we address the safety issue in pervasive space, by focusing one special element: device. We seek answers to two fundamental questions about safety from a device-oriented perspective. *First, how does a device understand, and inform of the notion of safety?* With each device in the pervasive space carrying its own safety knowledge and concerns, the notion of safety is constituted by scattered, sometimes conflicting pieces of information. Presenting an integrated and consolidated view of safety based on the collective knowledge from all stakeholders requires a uniform semantic framework allowing devices to express, exchange and share these safety knowledge and concerns using domain-independent concepts. *Second, how can we make devices participate proactively and intelligently to respond to unsafe scenarios?* Existing safety and security applications control devices by following explicit rules and policies. For example, alarm is triggered when the house is broken into; or sprinkler is activated if a fire is detected. These are rules pre-defined or hardcoded in the application logic. However, asking users and programmers to specify rules for each and every potential risk scenarios is not a scalable approach. It would be more desirable for systems to automatically enact devices to mitigate and eliminate the risky context. For example, when a door is left open at night, a system should be intelligent enough to

discover the door actuator and invoke the device to close the door. This ability to discover and invoke device services requires each device to effectively communicate its capability and identify its link with the risky context.

This chapter reports on our research efforts to answer the above questions. We start by a brief introduction to the Device Description Language (DDL), which is a base language to which we extend with safety features. Then we show our approach to the first question by introducing several DiOS enabled language constructs, which allow device manufacturers to specify their safety concerns and knowledge in universal terms within a uniform language structure. With DiOS and DDL, device manufacturers are able to express their concerns and knowledge in universal safety concepts within a uniform language structure. To answer the second question, we first present a device model that reveals a sensor-actuation link using the concept of context. Inspired by this sensor-actuation link, we then bring a third component to our language framework: a domain ontology for sensor taxonomy (DoST). DoST allows each device to inform of its domain of operation, which is an important clue for pervasive systems to discover and coordinate proper devices to cope with various safety risk scenarios. The chapter closes by surveying the related work and summarizing our own contributions.

Introducing DDL: the Device Description Language

DDL is an XML-based markup language that allows device manufacturers to publish the specifications of their products in a digital form. DDL covers a wide range of sensors and actuators. Each DDL device descriptor contains: (a) meta-information of a device for service registration and lookup, and (b) descriptions of communication interfaces between a device and external services. By providing a language processor that automatically converts a DDL device descriptor to a service bundle, responsibilities

among device manufacturers, system integrators and applications programmers can be distributed and clearly separated [1]. For each device, its manufacturer will provide the DDL descriptor, essentially a digital “datasheet” that specifies its interface; a system integrator can then run the ATLAS-DDL language processor [24] to create an OSGi bundle for the device, and upload it to the device repository to make the service readily available for application programmers. After a device bundle is created, it is uploaded to the Atlas Bundle Repository. Once a device connected to an Atlas node powers up and joins the network, the Atlas middleware, which listens on a variety of network interfaces, will identify the device and access the Bundle Repository to retrieve the service bundle. When a reference for such bundle is loaded, applications and other services are able to dynamically discover and access it using OSGi provided mechanisms.

DDL was originally designed and developed to enable automatic integration of a variety of devices [ref]. As our research proceeds, we repurposed DDL as a carrier of safety concerns and knowledge for devices. In this paper, we focus on the safety extensions to the language. We present more details of DDL in Appendixes: the DDL data model in Appendix A, the DDL schema in Appendix B, and a sample DDL descriptor file for a temperature sensor in Appendix C. The complete manual of DDL and the source code of the language processor have been released and are accessible online at http://www.icta.ufl.edu/atlas/ddl.

Expressing Safety Knowledge and Concerns

Device safety is an integral part of the overall safety of pervasive spaces. Many sensors and actuators deployed in the Internet of Things are primitive, pin-head sized chips that are vulnerable from harsh conditions and physical damages. For other devices, some built-in protection mechanisms are in place. However, they usually do

not cover every risk scenarios and cannot guarantee the total safety of the devices in use. Most devices still require users to be conscious of its vulnerability at the time of deployment and operation. We identify the following four categories of risk factors that devices could be exposed to in pervasive spaces.

- *Hostile environment*. The physical limitations require most devices to deployed and operated in a friendly environment. Harsh conditions such as extreme temperature, humidity, radiation, high levels of chemical substances, motion and vibration could compromise a device's function and jeopardizes its safety.

- *Interference*. Existence of some devices could cause interferences to others, and affect their normal operations. Interferences are considered hazardous to mission-critical devices. For example, airplanes ban the use of cell phones to avoid interferences to avionic devices.

- *Misuse*. To ensure the proper operation, many devices require their external users to follow certain procedures, which define the permissible sequence of interactions, and the boundaries of acceptable parameters. Violating the procedure could cause a device to enter a wrongful and unsafe state. For example, an application that repeatedly invokes a door actuator at 100 times per minute could break the actuator when the device no longer withstands the metallic fatigue.

- *Internal failures*. Devices could fail due to internal software or hardware errors. Some failures are critical, which could compromise the safety of the device itself and the space as well. For example, an ill-programmed firmware on a sensor node could cause self-heating, and eventually burns the node. Although these failures cannot be prevented by external precautions, they needed to managed and controlled in a timely fashion to avoid propagation and escalation.

Protecting devices from the above safety risks requires deep and detailed knowledge of devices from system integrators, service developers and users, who will deploy, program and interact with these devices at different stages of the pervasive system. Clearly, safety has put on extra burden to all above stakeholders as they have to go over a variety of manuals, specifications, and safety codebooks to collect these knowledge. Not to mention that safety rules and policies written in natural language are often vague, inaccurate and inconsistent.

Our approach redistributes the responsibility for safety by engaging one more stakeholder: the device manufacturer. Device manufacturers possess the most in-depth knowledge about their products. Hence it is natural for them to assume the role of specifying the safety needs of devices. To further extend their power, we provide a device description language that allows the specification of safety constraints in a uniform structure, and a tool that automatically converts device descriptors into software components that oversees the safety of the device. In the remainder of this section, we first introduce DiOS, a minimum ontology that defines the core safety vocabularies. Then we present DDL, the device description language and its implementation that utilizes DiOS vocabularies for representation and integration of device safety knowledge.

The following elements are added to the DDL schema to enable safety support in the language.

Context Constraint

Context constraint is a syntactic structure allowing DDL descriptors to specify safe (or unsafe) context for devices. Context constraints are mostly used to describe environment requirement of devices such as operating temperature and humidity. The structure of a context constraint includes a *context description* and a *safety classifier*. Context description specifies the context entity to be constrained, and the range (or set) of values for the context entity. DDL allows both built-in and user-defined context entities. The built-in context entities currently support four most common context domains: temperature, humidity, luminance and acceleration. To create a user-defined context entity, the user needs to specify the entity name, value space, unit, and sensor type associated to the context entity. The range (set) of values specified in context

description should be a subset of the value space, measured in defined units. The safety classifier is a predicate that asserts the safety level of the described context. DDL supports six classifiers as defined in DiOS vocabulary: permissible, impermissible, desirable, neutral, risky and severe. Figure 6-1 shows an example of a context constraint for an ambient light sensor using the built-in context entity temperature.

Conflict List

DDL allows each type of device to specify a list of conflicting device types with which it could interfere. There are two ways to specify a conflicting device. One can specify a device directly by name and model, or indirectly by domain of the device. Name and model attributes together, uniquely identify a device type. However, a device may conflict with a general type of devices with many individual models. Specifying devices by domain is therefore more convenient in these situations. A domain is a special attribute in DDL that classifies a device based on its field of operation. For example, we consider a heater and an air conditioner are potentially conflicting because they both operate in the domain of temperature. We will describe the concept of device domains in more detail when we introduce a domain-based taxonomy of devices in the next section.

Fail-safe Method

Fail-safe methods are actions to ensure a device cause no harm, or minimum harm to the space in the event of failure. Fail-safe methods vary upon devices, which may include shutdown, network cutoff, and recovery to a safe state, etc. Not every device has fail-safe features. To specify a fail-safe method in DDL, one needs to tag a device *method* with the optional "*fail-safe*" attribute. Figure 6-2 shows a DDL snippet of fail-safe methods for Type 3374-31 electric actuator. Type 3374-31 actuator is a linear

actuator commonly used with a valve in heating, ventilation and air-conditioning (HAVC) systems for buildings. The following code snippet specifies the actuator stem fully retracts (which keeps the valve open) as a fail-safe action.

In addition to above features, DDL also specifies operation protocols for devices, which regulate the interactions between a device and its users. DDL comes with a language processor, which converts DDL descriptors into software “driver” programs, what we call device service bundles. A device service bundle is a virtual representation of a physical device. To protect a device from misuse, a bundle provides a constrained service interface, which limits the sequences and frequency of method invocations, as well as the boundaries of parameters. Other devices and services can only interact with the device through its service interface. A device service bundle also contains the context constraints of the device. When a device powers up and joins the network, its corresponding service bundle will be activated and submit its context constraint to an integrated safety knowledge base. Detailed description of DDL language processor and device service bundles is presented in [1]. We presented only the extension in terms of concepts and specification (DDL schema as shown in Appendix B), but the current implementation of the DDL language processor has yet to support these extensions.

With DDL/DiOS, we are able to specify safety constraints for individual devices and collect them into an integrated safety knowledge base, which makes possible a global approach for safety: a single device in the pervasive space can rely on the overall sensing capability of the entire system to monitor for its undesired context, and the overall actuation capability to mitigate when risky context occurs. However, one

question remains: in the event of an impermissible context, how does the system know which proper device to actuate to avert/mitigate the context?

Engaging Devices in Risk Mitigation

To proactively engage devices in risk mitigation requires pervasive systems to understand the relation between safety risks and participating devices. In our approach, we model this relation by the concept of context. In the following section, we present an abstract model of devices that reveals the link between sensor, actuator and context.

Devices: a Context-Based View

We classify devices in pervasive systems into two basic types: a *sensor* that observes and an *actuator* that affects, a particular domain of the physical space. Most devices in the pervasive space can be considered as either a sensor, or an actuator, or a combination of both. As defined in DiOS, we use the term *context entity* to denote a "domain" of the pervasive space, and context to denote an instance in the domain. With these two concepts, we are able to link a sensor and an actuator together if they operate within the same domain. For example, we can say a thermostat (sensor) is a reader of temperature (context entity); and an AC a writer of temperature. When a undesirable scenario such as over heating arises, we can use a thermostat to monitor and detect, and an AC as a potential candidate to resolve this undesired context.

We define *context entity* as a conceptual entity that represents the domain of a sensor or an actuator [9]. A context entity has a value space and a metric. A *sensor* is a physical or virtual entity that performs *sensing*, which is a read operation over a context entity; an actuator is a physical or virtual entity that performs actuation, which is a write operation over a context entity. A more formal definition of the above concepts can be found in Chapter 6.

The concept of context entity changes our understanding of devices, which brings several benefits in addressing the issue of safety. First, it defines the scope of the safety problem for pervasive systems. While there are many safety risks and hazards in the pervasive space, not all of them are detectable by pervasive systems. The safety responsibility of pervasive systems is bounded by its sensing and actuation capabilities. A pervasive system should only be accountable for safety issues in domains accessible to the system; in other words, the safety conditions that can be represented in context, readable and modifiable by its sensors and actuators. Second, context enables an abstract yet accurate representation of safety concerns and knowledge. On one hand, existing safety rules and constraints represented in natural languages could be too abstract for pervasive systems to practice. On the other hand, modeling risky conditions and phenomenon directly by sensor readings is accurate but less re-usable. These models are often deployment-specific (e.g. depending on sensor locations, types and models, etc.) and not suitable for general safety knowledge and concerns. Context representation provides the right balance between abstraction and accuracy, which is ideal for describing safety knowledge. Third, context reveals the link between sensors and actuators, providing an important cue for discovering proper actuations to recover from unsafe context.

DoST: a Domain Ontology for Sensor Taxonomy

Pervasive systems should be capable to assign the proper sensors for detection, and the proper actuators for mitigation, when risky context arises in certain domains (i.e. context entities). If each device can specify its domain of operation, pervasive systems will have better clues to search and match devices with various risk scenarios. This requires a standard taxonomic scheme to classify and organize all domains of devices.

The Domain ontology for Sensor Taxonomy (DoST) is our attempt to define such taxonomic scheme.

While there are many types and models of sensors and actuators, we observe that there is only a limited set of domains within which they operate. Therefore, a classification of device domains is practically feasible. We create the DoST ontology by surveying most commonly available sensors in the market. Although DoST can also serve for actuators, we are only interested in sensor domains because: (1) actuator domains largely overlap with sensor domains; (2) safety risks are eventually represented in sensor readings. Actuators operating in domains invisible to sensors are not useful in risk mitigation.

As shown in Figure 6-3, DoST has a hierarchical structure, with class *ContextEntity* at its root. Every sensor domain is a context entity. DoST define four top-level domains: space, pervasive object, device, and user. Each of these domains is further divided into more sub-domains. For example, space domain includes weather, sound, luminance, flame, chemical level, etc. Pervasive object includes sub-domains such as identification, location, orientation, presence, size, speed, weight, etc.

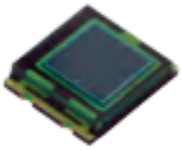
We incorporate DoST in DDL by allowing devices to specify their domains of operation in the DDL descriptor. With DoST, a pervasive system is more informed of the capabilities of its devices, can more effectively discover and enact devices to cope with risky conditions. Sensors and actuators, instead of being mandated by strict application logics, can now voluntarily participate in critical tasks for safety monitoring and risk mitigation.

Summary

Many existing pervasive applications focus on monitoring the safety of the environment [26][30], human [27] and infrastructures [28][29]. However, the safety of devices has remained a largely unexplored area. In addition, many existing safety applications extensively use sensors for safety monitoring; but only a few report the use of actuators to mitigate from risk conditions [31]. Moreover, all systems surveyed above have specific and predefined safety risks to protect against.

Our approach distinguishes itself from the related work cited above in the following ways. First, we do not assume or predefine specific safety risks. Instead, we provide a language framework allowing stakeholders to specific any safety concerns using a general context abstraction. Second, we address safety from a device-centric perspective. We find a device, as a point of interaction between the cyber space and physical world, could directly cause physical harms and damages. Restraining devices from wrongful and risky operations gives us the best chance to enhance the space safety. We also find safety of devices is an important aspect contributing to the overall safety of the space, which requires us to understand the limitations of devices and follow their operation procedures. Third, our approach utilizes actuators in addition to sensors. Pervasive systems possess the power to actively affect the physical space through actuations, which can be used to modify unsafe and impermissible context and mitigate from risk conditions. We aim to unleash this power by linking actuators with sensors using the concept of context.

```
<Sensor>
 <Description> …
   <Constraint>
    <ContextDescription entity="temperature">
     <Range>
      <Max>100</Max<Min>-40</Min>
     </Range>
     <Unit>Centigrade</Unit>
    </ContextDescription>
    <SafetyClassifier>Desirable</SafetyClassifier>
   </Constraint>…
 </Description>
 <Interface>…</Interface>
</Sensor>
```



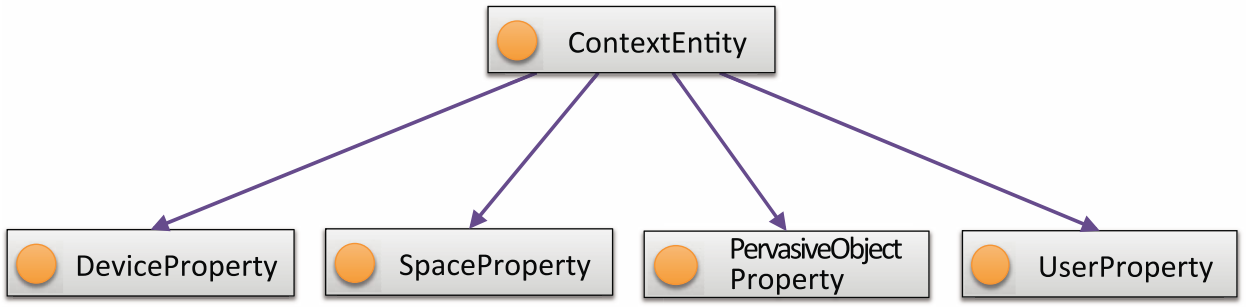
A TMD5510FX01 ambient light sensor manufactured by Vishay, Inc.

Figure 6-1. TMD5510FX01 ambient light sensor and its DDL snippets. (The snippets define the sensor support operating temperatures from -40 °C to +100 °C.)

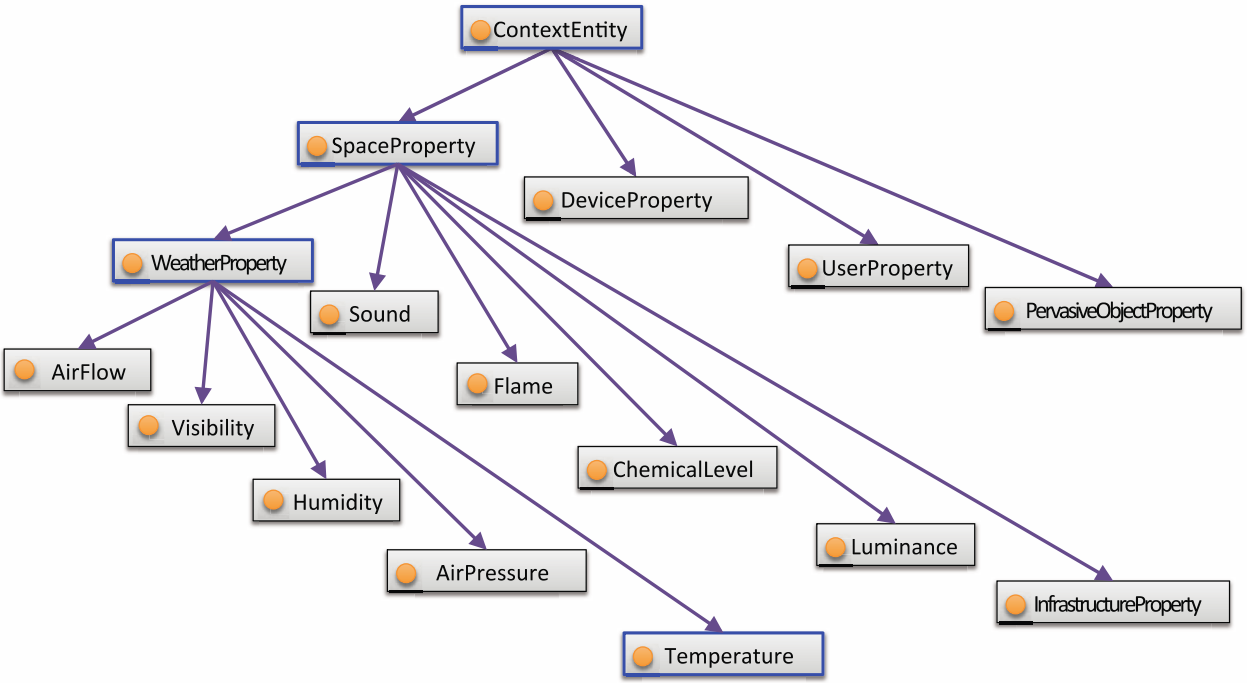
```
<Actuator>
 <Description>…</Description>
 <Interface> …
   <Method fail-safe="yes">
    <MethodName>retract</MethodName >
    <Parameter name="travel_distance" value="15"
unit="mm"></Parameter>
    <Parameter name="actuate_time" value="12"
unit="s"></Parameter>
    <Parameter name="positioning_force" value="500"
unit="N"></Parameter>
   </Method>…
 </Interface>
</Actuator>
```

A Type 3374 linear actuator attached to V2001 globe valve by Samson, Inc.

Figure 6-2. Type 3374-31 electric linear actuator and its DDL snippets. (The snippets define the fail-safe method “retract” for the actuator.)



(a) Top-level taxonomy of DoST



(b) Partial view of DoST showing the taxonomy branch for *temperature*.

Figure 6-3. A Snippet of the DoST ontology.

CHAPTER 7
FORMALIZING SAFETY IN PERVASIVE SPACES

Motivation

Pervasive systems bear some resemblance to database management systems. A database management system controls the use and maintenance of the database, which is an integrated collection of shared data objects. Transactions are executed on the database by performing read and write operations to access and manipulate these data objects. Similarly, a pervasive system is deployed in a certain space and maintains the physical state of the space, which is logically represented by a collection of *context* entities in the system. To access and control the state of the space, applications in a pervasive system invoke sensing and actuation services to perform reads and writes on various context entities.

In database, failures and concurrency are the two sources of potential errors that lead to inconsistent states of data. We find a similar class of problems in pervasive systems that are raised by the same factors. For example, a sensor failure could cause false readings of the context, leading to incorrect interpretations of physical state of the space (such as a false positive or true negative detection of a safety risk). An actuator failure could result in a failed attempt of modifying contexts, creating an unknown state of the space. In addition to device failures, service and applications could also fail due to logical errors and erroneous executions, giving rises to inconsistent and false states in the physical world. Even with faultless applications that are correctly executed, concurrent executions may also create scenarios of conflicts where applications are contending to access and change the same context. For example, one application wants to keeps the light off, while the other wants to kept it on. The light will be turned

on and off repeatedly, which causes a volatile and undesired state of the space. We generalize this class of problems by the term of safety problems in pervasive spaces.

Despite the similarities between the two types of systems, distinctions should not be neglected.

First, data stored in databases are *digital*, while context describing the physical properties of the space, in some sense, can be *analog*. In database, the value of a data object takes discrete states. A write operation changes the value of data from one state to the other. There are no intermediate states in this operation. However, in pervasive spaces, many physical properties are continuous. For example, actuating the AC to change the room temperature from 85 degrees to 75 degrees is a continuous process in which the state of the temperature context takes a gradual change. Therefore, the first implication of this analog property of the pervasive space is that there exists a *temporal relationship* between actuations (the operation) and context entities (the operand). While write is simply considered as an assignment of value in database systems, actuation can be modeled as a temporal function of context entities in pervasive spaces. The second implication is *fuzziness*. Data objects in database system take exact values. But in many cases, it is practically infeasible to maintain the same exactness when we describe the physical properties of a pervasive space. Following the previous example, if the AC manages to settle the room temperature at 74.9 degrees, do we really consider the actuation has failed? Clearly we need to introduce the concept of fuzziness to context descriptions for pervasive systems.

Secondly, databases are *closed* systems where only read and write operations from the database management systems are allowed to access and manipulate data.

Pervasive systems, in contrast, are usually deployed in a space *open* to external factors such as environment effects and human activities. Unlike database, pervasive systems do not have the exclusive access to modify the state of context entities. Physical dynamics and human activities in the space can also bring in uncontrollable, and sometimes unpredictable, modifications to the context.

Last but not least, database systems and pervasive system have very different goals. The primary concerns of database are the integrity and consistency of data protected by properties known as ACID (atomicity, consistency, isolation and durability), while pervasive spaces typically aim to maintain a safe and desirable state of the space by carrying a number of explicit or inexplicit constraints. Although we have identified that both pervasive systems and database shared a mutual class of risk factors such as failures and concurrency, it appears that the concerns and criterions of both systems are not equal. For example, actuating an AC in the room while reading from a temperature sensor at the same time would be considered as “dirty read” in the sense of traditional database. But should it really be a concern in the pervasive system? While it is strongly suggested that pervasive systems bear some transactional nature, the original ACID properties are not suitable to describe this nature and therefore need a revisit.

Revisiting ACID properties for Pervasive Systems

ACID is the set of properties that regulate the processing of transactions in database management systems. Here, we extend the concept of transaction to pervasive spaces, to represent a unit of execution of pervasive services, which consists of a number of sensing and actuations, similar to read and write operations in

databases. Formal definitions of transaction, sensing, and actuation can be found in the following sections.

Atomicity

In database, atomicity means that either all or none of the transaction's operations are performed. Therefore, all operations of a transaction are treated as a single, indivisible, atomic unit [12]. Atomicity rules out the possibility for incomplete transactions in database: a transaction is either processed to its entirety or not processed at all. Scenario B in Section 2 demonstrates a case where the non-atomic execution of services creates an impermissible intermediate states. However, the violation of atomicity does not always lead to compromise of safety. It is okay to cancel a transaction in the middle of its execution as long as the resulted intermediate state is safe. In addition, atomicity implies the ability to rollback when a transaction cannot proceed. While it is easy to cancel the effect of write operations on a data object, it is much more difficult and costly to rollback an actuation that changes the physical state of a space. In many cases, it requires compensating actuation operations to restore the space to its previous state. Therefore, the property of atomicity requires a relaxed definition for the new purpose of safety.

Consistency

Consistency requires that a transaction maintain the integrity constraints on the database. These integrity constraints have been largely relaxed in pervasive spaces. Most pervasive systems can tolerate with inconsistency issues such as "dirty read" and "lost update". Here we propose to replace consistency with a new property of integrity. Slightly different than its original meaning in database, integrity of a pervasive space indicates that each entity in the space always remains in a safe state. A pervasive

system protects integrity of the space by eliminating transactions that could lead to a undesired or risky state (we call impermissible context).

Isolation

Isolation demands that a transaction execute (as though) without any interference from other concurrent transactions. Same principle holds true for pervasive spaces: ideally, a pervasive application should not worry about others to ensure its correct execution. Isolation also implies exclusiveness. A transaction in execution has exclusive access to the context entities it modifies. But this is much more difficult to achieve in pervasive spaces, where external factors such as environment changes and human activities could always interfere with its execution. While it is unreasonable and practically impossible to isolate pervasive systems from the external world, at lease some level of isolation can be achieved by separating conflicting services within the system.

Durability

In database, durability means that all the changes made by a successfully terminated (committed) transaction become permanent, surviving any subsequent failures. To achieve such persistence of data, database management systems typically employ replications and recovery techniques. Unlike database, which always attempts to recover to a particular point in its state space, a pervasive system only requires the persistence of subspace, which is a set of permissible states of the space. When failures and natural hazards take place, a pervasive system will try its best effort to recover the space to any permissible state. To summarize, the meaning of durability in pervasive space has largely changed: while the effects of applications do not need to be permanent, the pervasive space desires a lower degree of persistence by trying to

survive from failures and always restore itself to a safe point of its state space, we call permissible context.

Context, Context Entities, and Context Envelope

The notion of context for pervasive space has been extensively discussed in the literature. However, none of the existing definitions adequately serve the purpose to define our transaction model. In this section, we present our own definition of context and introduce an associated concept: context entity.

In a pervasive space, sensors perceive the state of the physical world. Each sensor produces sensor readings, describing a certain aspect of the physical world. All sensors in the space together, define the perceived state space of physical world. We call this perceived state space “the context space” of a pervasive environment. A *context*, is simply a point in this context space.

Context and Context Entities

We introduce the notion of *context entity* as a conceptual entity to refer to a “subject” that a sensor is sensing. Context entity may not have a material existence. For example, room temperature becomes a context entity if there exists a temperature sensor available in the space. If we model sensing as a “read” operation, its corresponding context entity becomes the operand that the sensor is reading upon. Similarly, we can model actuations as “write” operations. A sensor reading could be affected by multiple actuations. Hence a context entity is associated with only one read operation, but 0 to many write operations. Context entity brings a powerful notion to our work because it changes our perspective, as illustrated in Figure 7-1, and reveals a new link between sensors and actuators: they are reading and writing to a shared set of

“entities”, whose states could be protected or locked by proper control of sensor and actuators.

Formal definitions of the concepts described above are presented in Definitions 7-1 to 7-3.

Definition 7-1. Context Space. For a sensor S_i, in the sensor set $\boldsymbol{S} = \{S_1, S_2, \ldots, S_n\}$ of a pervasive space $\boldsymbol{P}$, the readings of S_i are of domain D_i. The context space of $\boldsymbol{P}$ is defined as a vector space $\boldsymbol{C^P} = D_1 \times D_2 \times \ldots \times D_n$.

Definition 7-2. Context and Context Vector. A context is a point in space $\boldsymbol{C^P}$, represented by an n-tuple: $c = (d_1, d_2, \ldots, d_n) \in \boldsymbol{C^P}$, where $d_i \in D_i$. Given two contexts $c_1 \in \boldsymbol{C^P}$ and $c_2 \in \boldsymbol{C^P}$, we define a context vector $v = \langle c_1, c_2 \rangle$, representing a transformation of context from c_1 to c_2. The projection of context vector v on domain D_i is represented by $v \bullet d_i$, where d_i is the unit vector of domain D_i. Two context vectors v_1 and v_2 are of opposite directions, if $\exists\ D_i$, such that $v_1 \bullet d_i$ and $v_2 \bullet d_i$ are also of opposite directions.

Definition 7-3. Context Entity. A context entity e_i is defined by a 2-tuple: $e_i = (S_i, \boldsymbol{A})$, where S_i is the sensor that senses the state of the entity, $\boldsymbol{A}$ is the set of actuators that could change the state of the entity. $\boldsymbol{A}$ can be an empty set. The state space of e_i is defined by the domain of S_i: D_i. For a context space $\boldsymbol{C^P}$, the set of all context entities is denoted as $\boldsymbol{E^P}$.

In the above definitions, we assume each sensor has exactly one domain of sensing. In practice, when there is a device with multiple domains of sensing, we can split this device into multiple logical sensors. For example, a blood pressure monitoring device monitors both systolic and diastolic pressure, which can be represented by two logical sensors: systolic pressure sensor and diastolic pressure sensor. We also

assume the domain of each sensor is a *totally ordered set*: $\{D_i, <_i\}$, where D_i is the set of all possible readings, and $<_i$ is the ordering relation. This assumption is natural since most sensors produce numeric readings or readings convertible to numeric values, *i.e.* domain $D_i \subseteq \mathbf{R}$, where **R** is the set of real numbers and totally ordered. A few exceptions are mainly signal-recording sensors (such as image and acoustic sensors), where we have to manually impose the ordering for their readings.

Context Envelope

One way to understand context is by using the analogy of "snapshots". A single piece of context is a snapshot of the space, recording the state of the pervasive space at a particular point of time. Imagine a space expert examining through a deck of these snapshots. If the snapshot demonstrates a safe and pleasant state of the space, he will label it as "desirable"; if the snapshot shows a risky and unsafe state, it will be labeled as "impermissible"; if the expert has no opinion or judgment over the snapshot, the label will be "neutral". Eventually, we can sort these snapshots into three envelopes based on their labeling. If the deck of snapshots exhausts all possible states of the pervasive space, then we derive a division of the context space into three sub-spaces: impermissible context space (***I***), desirable context space (***D***) and neutral context space (***N***). This division of context space is called a *context envelope* (Definition 7-4). Figure 7-2.a shows an example of context envelope that divides a space with three context entities: temperature, humidity and luminance. Using this context envelope, given any context, we can quickly determine its safety implications. Note that ***D***, ***I***, ***N*** are not necessarily continuous spaces. For example, there could be several disconnected

“islands” of $\boldsymbol{D}$ representing the several desirable context spaces in our visualization of the context envelope (Figure 7-2.b).

A context envelope is a contextual specification of safety knowledge and concerns. While these safety knowledge may originate from multiple stakeholders such as user, domain experts and application programmers, and are represented in various forms such as safety documents, user profiles, and device manuals [11], context envelope provides a uniform representation of all these safety constraints and preferences that can be managed and interpreted by the pervasive system. The procedure to generate and maintain a context envelope is similar to the “snapshot” example described previously. A pervasive system at runtime maintains a global running context envelope. The context envelope is updated when a new safety constraint is inserted, or an existing constraint is removed. Any stakeholder can specify a safety constraint as a target set of contexts with a label of either “I” or “D”, using a simple domain-independent safety ontology.

Definition 7-4. Context Envelope. The context envelope of $\boldsymbol{C^P}$, denoted by $\varepsilon(\boldsymbol{C^P}) = \{\boldsymbol{I}, \boldsymbol{D}, \boldsymbol{N}\}$, is a division of $\boldsymbol{C^P}$ into three sub-spaces: impermissible context space $\boldsymbol{I}$, desirable context space $\boldsymbol{D}$ (the notation of desirable context space $\boldsymbol{D}$ is in bold font to be distinguishable from the domain of a context entity D), Neutral context space $\boldsymbol{N}$, such that:

1. $\boldsymbol{N} \cup \boldsymbol{I} \cup \boldsymbol{D} = \boldsymbol{C^P}$; and

2. $\boldsymbol{I} \cap \boldsymbol{N} = \emptyset$, $\boldsymbol{D} \cap \boldsymbol{N} = \emptyset$, and $\boldsymbol{I} \cap \boldsymbol{D} = \emptyset$

If a context $c \in \boldsymbol{N} \cup \boldsymbol{D}$, we say c is *in* context envelope ε; otherwise c is *outside* the context envelope.

The global context envelope changes over time. Arranging these varying envelopes along the time axis enables an interesting perspective, what we call the tunnel-view of context envelopes (Figure 7-3). A tunnel is a set of overlaid context envelopes that sets the boundary between context spaces over time. For example, in Figure 7-3, the vertical dashed lines outline the boundary between ***I*** and ***N*** $\cup$ ***D***. We name the outer part of the boundary ***I***-tunnel, and the inner part ***N***, ***D***-tunnel. Therefore, the goal of our safety protocol is to ensure the state transition of pervasive space (as indicated in bold arrows) stays within ***D***-tunnels as much as possible while avoid penetrating into ***I***-tunnels.

Sensing and Actuations

In our model, sensing and actuation are the only operations that allow a pervasive system to access and control the physical space, or logically "read" and "write" to various context entities. A sensing operation reads a single context entity by invoking a sensor to take a measurement, and returns the sensor reading as the result of the operation (Definition 7-4). The case for actuations is slightly more complicated, as a single actuation may involve multiple context entities. Consider an actuation *ActuateDoor*. Assume there exists a door actuator service that opens and closes a door, and a few sensors that read the following context entities: *door_status*, *temperature*, and *smoke_density*. We can define the *ActuateDoor* actuation as a write operation over *door_status*. Once such actuation is invoked, its effect over the *door_status entity* is clear and predictable. For example, if the *ActuateDoor* command with a parameter "open", is issued to the underlying device, we would anticipate the value of *door_status* will be set to "open" – unless failure occurs. According to our common knowledge, the

other two context entities, *temperature* and *smoke_density*, may also be affected, although in a more obscure and less predictable way. We use the term "affect" to describe this indirect relation between actuations and context entities, and "write" for the direct relations.

Definition 7-4. Sensing Operation. A sensing operation, denoted by $r(e)$, is defined over a context entity e, and returns the reading of the single sensor associated with e.

Definition 7-5. Actuation Operation. An actuation operation, denoted by $w(e, d)$, is defined over a context entity e, and a parameter $d \in D_e$. e is called the *written entity* of operation w, and d is the *intended value* to be written to e. A sensing operation $r(e)$ returns d upon the completion of $w(e, d)$, if w succeeds. In addition to changing the written entity e, an actuation w may also affect other context entities, what we call the set of *affected entities* $A_w = \{e_i \mid e_i \neq e \text{ and } e_i \in \boldsymbol{E^P}\}$. The relation between actuation w and its affected entity e_i, is described an *affect function*, as defined in Definition 7-6.

Definition 7-6. Affect Function. An affect function φ_i over context entity e_i, is a probability distribution function over domain D_i. Given an actuation w with an affected entity $e_i \in A_w$, we use $\varphi_i(d)$ to denote the probability that $r(e_i) = d$, $d \in D_i$, upon the completion of w.

Hence, in context space $\boldsymbol{C^P}$, a completed actuation w is a transformation of context, represented by a context vector $w = \langle c_{pre}, c_{post} \rangle$, where c_{pre} , $c_{post} \in \boldsymbol{C^P}$.

Affect functions allow us to explore the hidden relations between actuations and context, and use actuators to indirectly affect our context space, which empowers our safety mechanism in two ways: first, the pervasive system has more complete

knowledge to determine the safety implication of a requested actuation; secondly, in risky and unsafe situations, the pervasive system has more options to escape from impermissible contexts by exerting effects indirectly through an actuation. Although deriving a useful and accurate affect functions is not the focus of this paper, we expect two possible approaches to be utilized: (1) *system configuration* that allows programmers and space administrators to manually specify the function through common knowledge and observations; (2) *system learning* that relies on the system itself to learn the effects of actuations through the history of sensor readings. Three things can be learned to refine the knowledge of an actuation:

- *Affected Entities*. Sensor readings are indications of whether a certain relationship exists between a sensor and an actuator, which help to learn the set of affected entities of an actuation.
- *Unit Vector*. If a relationship exists between a sensor and actuator pair, observing the direction of sensor reading changes would help derive the unit vector for the context transformation.
- *Probability Distribution Function*. The ultimate learning possible is to derive the probability distribution function that indicates how likely a context will be achieved after an actuation is executed.

Pervasive Transactions

A pervasive transaction comprises a unit of work performed in pervasive systems. The basic components of pervasive transactions are sensing and actuation operations, which operates within the context space of the pervasive system. The goal of pervasive transaction is to maintain the safety of the pervasive system.

A pervasive transaction represents a transformation of context mutually agreed on by the transaction requestor (typically a pervasive application) and the executor (the pervasive system). A pervasive transaction *commits* (denoted by *CMT*) if the promised context is delivered; and it *aborts* on failures. Unlike database systems, which mandate

invalidation of all updates on an aborted transaction, pervasive systems are more tolerant with inconsistency issues, and do not require an exact invalidation. However, failed transactions may result in undesired and unsafe situations. To ensure the space stays at a permissible context when the transaction aborts, the system needs to enact proper transactions to recover the context to a safe range – a process we call *fail-safe abort* (denoted by *FSA*). The formal definition of pervasive transaction is presented in Definition 7-7.

Definition 7-7. Pervasive Transaction. A pervasive transaction T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r[e_j], w[E_k, E_m] \mid e_j \in \boldsymbol{E^P}, E_k \subseteq \boldsymbol{E^P}, E_m \subseteq \boldsymbol{E^P}\} \cup \{FSA_i, CMT_i\}$;

2. $FSA_i \in T_i$ iff $CMT_i \notin T_i$;

3. if t is FSA_i or CMT_i, for any other operation $p \in T_i$, $p <_i t$; and

4. if $r[e], w[E] \in T_i$, then either $r[e] <_i w[E]$ or $w[E] <_i r[e]$.

The above definition of pervasive transaction is almost a direct translation of transaction from the conventional database definition. However, coming into the realm of pervasive computing, this definition suffers from insufficiency of expressiveness in the following aspects:

First, in pervasive spaces, typically a transaction is issued to with the intention to achieve a certain context. Even if all operations inside a transaction are successfully carried out, it is still possible that the expected context is not delivered due to interferences from natural events and human factors. Therefore committing a transaction should not be depending on the completion of operations, but rather the

fulfillment of intentions. The current structure of pervasive transaction cannot express this notion of intention.

Secondly, many pervasive applications are time-sensitive, which means if the underlying transactions fails to deliver the intended context within a desired time-window, the system should also abort the transaction.

Last, many pervasive application not only wishes its intended context to be realized, but also persist for a while. Consider a sleep application that closes the window before resident goes to sleep. If another application re-opens the window immediately after the completion of the sleep application, it would defeat the purpose of the sleep application entirely, despite its successful completion.

Definition 7-8. Improved Definition of Pervasive Transaction. A pervasive transaction is represented by a four-tuple: $PT_i = (T_i, \varepsilon_i, TO_i, TTL_i)$, T_i is the partial ordering of operations as defined in Definitions 7-4 and 7-5; ε_i is the target context envelope of the transaction; TO_i is the time-out window for the transaction to enter ε_i upon the completion of its operations; and TTL_i is the *time-to-live ticket* for the transaction to remain within ε_i after commit.

One benefit of introducing the concept of transaction into pervasive systems is that it enforces isolation to avoid conflicting operations during execution. However, the notion of conflict requires re-thinking when we consider the concurrent nature of pervasive systems. As we shift our focus from data/context consistency to safety in pervasive spaces, our transaction model should define a more cope-able criterion for isolation that preserves the way pervasive systems operates, as well as eliminates those operations that raise safety risks. Our model provides two levels of isolation: the

conflict-based isolation that prevents simultaneous actuations on common context entities, and the compatibility-based isolation that separates transactions based on "conflict of intentions". The compatibility-based isolation provides a more relaxed concurrency control without significantly compromising the safety of pervasive spaces. Definition 7-9 and 7-10 define conflict and compatibility respectively, and the concept of isolation is presented in Definition 7-12.

Definition 7-9. Conflict. Two operations are in conflict with each other, if

1. both operations are actuations, denoted by $w_1[E_1]$ and $w_2[E_2]$; and
2. their write sets contain common context entities: $E_1 \cap E_2 \neq \emptyset$.

Two transactions T_1 and T_2 are in conflict with each other, if $\exists\ w_1 \in T_1$ and $w_2 \in T_2$ such that w_1 and w_2 are in conflict.

Definition 7-10. Compatibility. Two write operations $w_1 = <c_{pre1}, c_{post1}>$ and $w_2 = <c_{pre2}, c_{post2}>$ are incompatible, if vectors $< c_{pre1}, c_{post1}>$ and $<c_{pre2}, c_{post2}>$ are of opposite directions.

Two transactions T_1 and T_2 are incompatible with each other, if $\exists\ w_1, w_2, w_1 \in T_1$ and $w_2 \in T_2$ such that w_1 and w_2 are incompatible.

AI^{2}D-safety

AI^{2}D Properties

Definition 7-11. Atomicity. A pervasive transaction is atomic if it either commits or fail-safe aborts at the end of execution.

Definition 7-12. Integrity. Integrity is violated if current context c is outside the context envelope $\boldsymbol{\varepsilon}$, i.e. $c \in \boldsymbol{I}$. More specifically, a pervasive transaction T_i satisfies the

integrity property if the post-conditions of all actuation operations of T_i are not impermissible: $\forall\ w \in T_i$, such that for w: $c_{pre} \rightarrow c_{post}$, $c_{post} \in \boldsymbol{D} \cup \boldsymbol{N}$.

Definition 7-13. Isolation. A pervasive system satisfies (a) *conflict-based isolation* if there exists no concurrent conflicting transactions during execution, and (b) *compatibility-based isolation* if there exists no concurrent incompatible transactions during execution.

Definition 7-14. Durability. A context space transaction satisfies the property of durability if its intended context survives the time-to-live period.

Define Safety Using AI^{2}D Properties

Based on the transactional model presented above, we give our definition of AI^{2}D safety in Definition 7-15.

Definition 7-15. AI^{2}D safety. A pervasive system is *AI^{2}D-safe* if the executions of all pervasive transactions within the system satisfy the properties of atomicity, integrity, isolation and durability.

Note that our definition is different from the common notion of safety. By saying a pervasive system is “safe”, we mean the pervasive system is acting safe within its own capabilities:

(1) the pervasive system will prevent all self-incurred safety risks; and

(2) the pervasive system recovers from a safety risk only if such risk is detectable

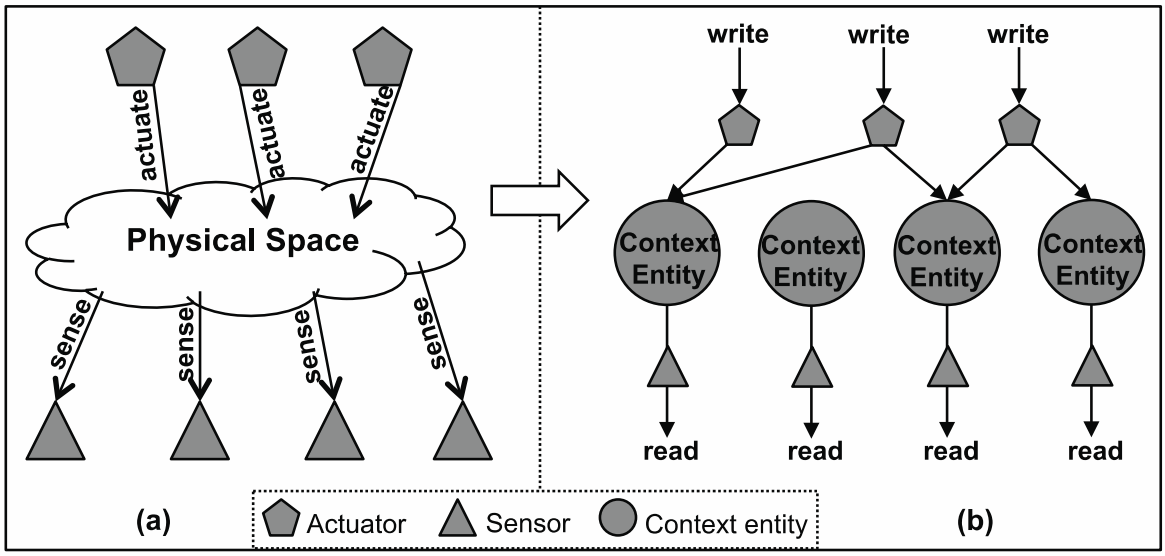


Figure 7-1. The concept of context entity enpowers a new perspective of sensor-actuator relations in pervasive spaces.

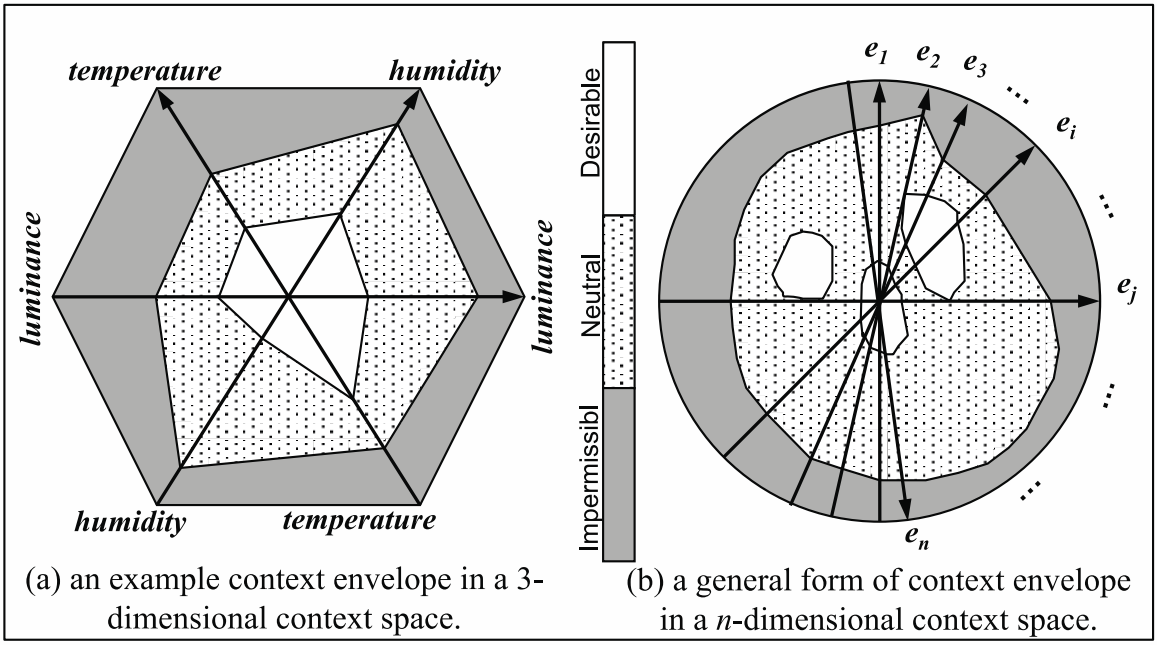


Figure 7-2. Visualization of context envelopes in 3-Dimensional (Fig. 7-2.a) and *n*-Dimensional (Fig. 7-2.b) context spaces.

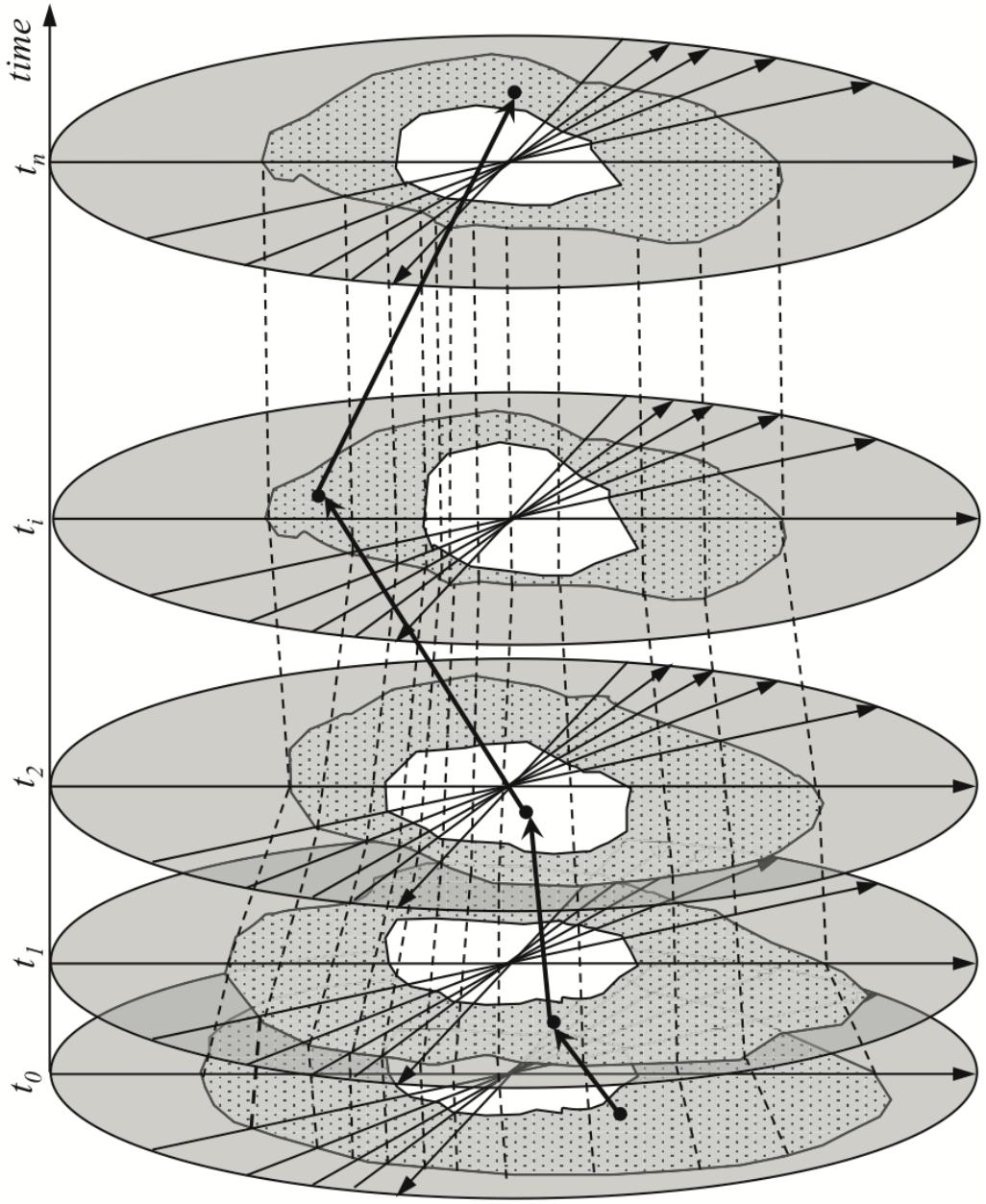


Figure 7-3. The tunnel-view of context envelopes.

CHAPTER 8
PERVASIVE TRANSACTION IMPLEMENTATION

Introduction

The concept of pervasive transaction enables a fresh perspective of safety management in pervasive systems. In this perspective, the interactions between pervasive systems and physical spaces are abstracted as pervasive transactions. A pervasive transaction is unit of execution consisting a number of sensing and actuation operations over context. The overall safety requirement of a pervasive space is abstracted as a context envelope. To maintain the pervasive space within the context envelope at runtime, we need to not only regulate and manage the behavior of pervasive transactions, but also proactively enact, what we call safety transactions, to recover from impermissible context. The AI^2D (atomicity, integrity, isolation and durability) properties together, provides a clear definition of the safety criteria, which serve as guidelines for the execution of pervasive transactions.

This chapter describes the implementation of pervasive transactions. We first introduce the runtime management of pervasive transactions, by presenting the state diagram of a pervasive transaction. We explain each state in the diagram and present state transition rules. We then zoom into two stages of pervasive transaction's life cycle: FAIL_SAFE ABORT and EXECUTE. To implement the idea of fail-safe abort, we introduce safety transactions, which is a recovery technique by utilizing existing sensing and actuation capabilities of pervasive systems. Then we introduce the context lock protocol, which regulates the execution of pervasive transactions by a variation of lock concept, which we call context locks.

Pervasive Transaction Model

In this section, we present a full model of pervasive transactions. We aim to address the following aspects of pervasive transactions in our model: (1) lifecycle, which consists of several different stages spanning from its creation to termination (abort or commit); (2) formal structure, which defines the necessary operations and parameters needed to formulate a pervasive transaction; (3) programming interface, which presents the tools for programmers to submit pervasive transactions.

Pervasive Transaction Overview

A pervasive transaction (also referred to as “transaction” in the remainder of the chapter) is a unit of execution in a pervasive system. It consists of a sequence of read and write operations defined over context entities, to represent the interactions between the cyber system and physical space. Pervasive transactions are issued by various applications and system services (referred to as “applications”), and then processed by a pervasive transaction manager (referred to as “TM”). To compose a transaction, a programmer is provided with a transaction interface (as a library in a programming language), which allows the programmer to assemble and customize available operations into transactions and embed them in the application program.

Pervasive Transaction Lifecycle

The pervasive transaction theory, at its core, is a model of the interaction between pervasive systems and physical spaces based on the concept of pervasive transaction. A pervasive transaction, as defined in Chapter 7, is a unit of execution within the pervasive system, which consists of a number of sensing and actuation operations. In this section, we describe the run-time management of pervasive transactions.

As shown in Figure 8-1, the lifecycle of a pervasive transaction is divided into six stages: INITIALIZE, SENSE_OR_ACTUATE, EXECUTE, PRE-COMMIT, COMMIT, and FAIL-SAFE ABORT. We introduce each stage in the following sections, and present a summary of state transition rule in Table 8-1.

INITIALIZE: the starting state of transaction management. At this stage, a pervasive transaction enters the transaction processor, which prepares its execution by assigning the needed resources to the transaction. After the INITIALIZE state, a pervasive transaction starts its normal operation in a two-state loop between SENSE_OR_ACTUATE and EXECUTE.

SENSE_OR_ACTUATE: the preparation state for a pervasive transaction to perform its next operation, which is either a sensing or actuation. At this state, the transaction requests all context locks needed for its operation to gain accesses to the relevant context entities (LCK_REQ). If all context locks are granted (LCK_GRT), the transaction proceeds to the EXCUTE state. If any of the context locks are rejected by the context lock manager (LCK_REJ), the transaction declares its failure by entering the FAIL-SAFE ABORT state. When the transaction finishes the execution of all operations (TR_COMPLT), it enters the PRE-COMMIT state. The detailed protocol for context lock management will be presented in Section: Context Lock Protocol.

EXECUTE: the execution state where a single operation of the pervasive transaction is performed. At this state, the transaction processor performs a sensing or actuation operation by invoking its corresponding sensor or actuator service. If the execution succeeds (EXEC_S), the transaction returns to the SENSE_OR_ACTUATE state to prepare for its next operation. The transaction is forced into the FAIL-SAFE

ABORT state if its execution fails due to any of the following reasons: (1) failure of the underlying device services (EXEC_F), (2) revoke of a previously granted context lock (LCK_RVK), and (3) detection of impermissible context during the execution (EXEC_F).

PRE-COMMIT: the state where a pervasive transaction waits to commit after all its operations complete (TR_COMPLT). This state assigns a time-out window for the effects of the completed transaction operations to stabilize. If the stabilized context enters the target context envelope within time-out window (WAIT<TO), the transaction is considered successful and ready to commit (TR_CMT). Otherwise (WAIT≥TO), the transaction fails to deliver the expected context and enters the FAIL-SAFE ABORT state (TR_ABRT). We present the procedure of PRE-COMMIT in pseudo-code in Figure 8-2.

COMMIT: the state that indicates the successful completion of a pervasive transaction. A transaction enters the state only if (1) all its operations are successfully performed without failure, and (2) the context enters the target context envelope within the prescribed time-out window. When a pervasive transaction commits, it exits from the transaction processor. To protect the durability of the achieved effects by the committed transaction, a safety transaction will be composed and invoked.

FAIL-SAFE ABORT: the state that indicates the failure of the transaction processing. FAIL-SAFE ABORT and COMMIT are the only two ending states of transaction processing. FAIL-SAFE ABORT interrupts the normal execution of pervasive transaction, and could raise safety concerns. Therefore, this state is also a checkpoint of the overall safety of pervasive space. If impermissible context occurs in the event of fail-safe abort, a safety transaction is invoked to restore the physical space to a safe context. Figure 8-3 shows the pseudo-code of the fail-safe abort procedure.

Pervasive Transaction Operations

In our pervasive transaction model, we provide three types of operations over context entities to programmers: read, write and free. Read and write are similar to the corresponding database transactions, which access and update the operands (context entities), respectively. Free is a special operation for pervasive transactions that empowers programmers to free a certain operand from isolation control. This is an important feature of our pervasive transaction model because our model design in based on the premise that perfect isolation is neither necessary nor feasible in pervasive systems. Relaxation mechanisms such as “free” are useful to further utilize application knowledge and enable a more practical and efficient transaction processing protocol. We will introduce a few more relaxations in our definitions of read and write operations.

Definition 8-1. General Notation of Operations. A general form of operations in a transaction is $O_{ij}[e_k]$, where i is the index of the transaction, j is the index of the operation within transaction T_i, k is the index of the context entity.

Definition 8-2. View Dependency. For operations $O_{ij}[e_l]$ and $O_{ik}[e_m], (j < k)$ in transaction T_i, $O_{ik}[e_m]$ is view-dependent on $O_{ij}[e_l]$ if the view of e_l provided by $O_{ij}[e_l]$ affects $O_{ik}[e_m]$, we can also say $O_{ik}[e_m]$ is dependent on the view of e_l.

In our transaction model, an operation $O_{ij}[e_l]$ is view-dependent on another operation $O_{ik}[e_m]$ by default if $O_{ik}[e_m]$ precedes $O_{ij}[e_l]$ in transaction T_i, i.e., $j < k$. However, our model also allows applications to specify exceptions to this default rule using the free operation.

Definition 8-3. Free Operation. A free operation, denoted as $F_{ij}[e_k]$, signifies an explicit permission of transaction T_i to give up isolation protection over the context entity e_k (for example, through releasing the lock over e_k). With a $F_{ij}[e_k]$ operation, T_i declares that any following operation $O_{ih}[e_k], h > i$, is not view-dependent on e_k. T_i is willing to allow other transactions to access e_k. Once e_k is freed by transaction T_i, subsequent operations over the same context entity e_k are disallowed in T_i.

Definition 8-4. Read Operation. A read operation in transaction T_i, denoted as $R_{ij}[e_k]$, is defined over a context entity e_k, i and j are the transaction index and the operation index within the transaction respectively. A read operation returns a reading of the context entity.

As discussed previously, there may be operations following $R_{ij}[e_k]$ in T_i that are dependent on e_k. Therefore it is necessary to ensure the view of e_k (i.e., the reading returned by $R_{ij}[e_k]$) must be consistent with the reality when such dependency exists. When $e_k$is interfered by other transactions or external events and cause its value to fluctuate, the magnitude of such fluctuation should be bounded to avoid affecting T_i. We introduce a novel approach in which our pervasive transaction model allows for read operation to be defined under such fluctuations. We allow programmers to assign an optional fluctuation threshold parameter δ to the read operation as the boundary of fluctuation allowed on e_k as the programmer and application semantics see fit. When unspecified, the default value of δ is 0. The full notation of read operation is defined as $R_{ij}[e_k](\delta)$, which requires the read operation to satisfy the δ-constraint. The latter is formally defined as follows:

Definition 8-5. δ-constraint. At any point during the *effective period* (as defined in Definition 7-6) of $R_{ij}[e_k]$, the value of e should satisfies: $|e - c| < \delta$, c is the value of e at the time when $R_{ij}[e_k]$ starts to be processed.

Definition 8-6. Effective Period. The effective period of an operation $R_{ij}[e_k]$ is the time period during which there is at least one operation including $R_{ij}[e_k]$ itself in T_i that is dependent on e_k.

For the purpose of simplicity, we use shorthand notation $R[e]$ to represent a read operation in this document, indices and optional parameter δ appear in our notations only when they are relevant to the discussion.

As introduced in Chapter 7, a write operation is an abstraction of the actuation process on the context level. The simple form of a write operation is: $\mathrm{W_{ij}[e_k](d)}$, which represents an actuation with intended effect d on context entity $\mathrm{e_k}$, d is a value in the domain of $\mathrm{e_k}$, i.e., $\mathrm{d} \in \mathrm{D_k}$.

A close examination of the above definition brings a few questions.

First, using a specific value d to represent the intended effect assumes that the actuator has at least the same resolution as its corresponding sensor. This is not true for many devices. For instance, the resolutions of most air temperature sensors (0.1℃ or lower) are much higher those of most air-conditioning systems (1℃ or higher). Such mismatch in resolutions can create problems when writes are issued using d alone, e.g., consider writing temperature to 22.5℃ while there are only 22℃ and 23℃ on the dial of the AC. In addition to the degraded resolution, actuators often have limited precisions. A precise write to a specific intended value d would be infeasible for many actuators. To accommodate the need for more realistic actuations, write operations

should relax the requirement on intended context d by specifying a margin of *deviation* as allowed by the application.

Second, the current definition of write operations makes no assumption of the temporal effect of actuations. Many actuations in pervasive spaces are not instant (unlike write operations in database systems). It takes time for an actuator device to interact with the physical space and progress into its intended effect. The duration of an actuation does not necessarily stop when the actuator completes its action or motion. Consider a window opener as an actuator that manages in-door air temperature. There is usually a time delay between the device action (opening the window) and intended effect (temperature settling into the intended range). A practical definition of write operations should take into consideration such delayed effect of these actuations.

Allowing write with delayed effect reveals a large class of actuations: actuations through human interactions. For this class of actuations, the actuators are human, who are commanded or persuaded by the system to take actions to derive intended contexts and effects. The system initiates an actuation by providing commands or persuasion cues to a human user, who is expected to follow the actuation command by performing specific actions. An example of such actuation is a medicine dispenser/reminder, which notifies (say, by a voice message through a speaker) the user when it is time to take a pill. In this case, a medicine dispenser sensor could be utilized by a “medicine reminder” context entity to check if the patient has taken her medicine before issuing the reminder (actuation command) and in case not, to verify if and when the action is taken following the actuation command. Unlike physical actuators, humans act on their own will and are less predictable in completing these system-cued actuations. Tolerance with time-delay

or failure in human execution should be considered when we use write operation to model these interactions.

Third, as we discussed in Chapter 7, there is a desire for durability - the post effect of an actuation (i.e., the achieved context of a write operation). However, as discussed in Chapter 7, durability in the context of pervasive transactions is different from that in the context of databases, and is limited to an extent of time (e.g., minutes, hours, or until a specific context envelop is reached). To achieve durability, we assign a "durability window" for each write operation. After a write operation completes, its achieved context will be protected for the duration of the durability window.

In light of the above observations, we extend the definition of write operation with three optional parameters: σ for range relaxation, τ for temporal relaxation, θ for durability relaxation. A write operation is defined as following:

Definition 8-7. Write Operation. A write operation in transaction T_i is denoted as $W_{ij}[e_k](d, \sigma, \tau, \theta)$, where operand e_k is a context entity, i and j are the transaction index and the operation index within the transaction, respectively. Parameter d is the target value intended to be written to e_k. The rest of the parameters are defined as:

Definition 8-8. $\boldsymbol{\sigma}$ Parameter. Optional parameter σ is the deviation relaxation threshold that specifies the maximum tolerated margin of deviation (we call σ-margin) for the context derived by the write (*post-actuation context*). A write operation is considered *successfully completed* only if post-actuation context d' satisfies the σ-margin specification: *i.e.,* $d' \in [d - \sigma, d + \sigma]$.

Definition 8-9. $\boldsymbol{\tau}$ Parameter. Optional parameter, τ, is a temporal relaxation window (we call τ-window), which specifies the maximum completion time allowed for

the write operation. τ-window opens when the transaction processor starts processing the corresponding operation $W_{ij}[e_k]$, and closes when the elapsed time counts to τ. If a write operation cannot successfully complete within τ-window (due to either physical/human actuator failure or physical interferences), the write operation is considered failed and the corresponding transaction will be aborted.

Definition 8-10. $\boldsymbol{\theta}$ Parameter. Optional parameter θ is the durability relaxation window (we call θ-window), which specifies the minimal time required for the post-actuation context to endure. θ-window opens when the corresponding operation $W_{ij}[e_k]$ is successfully completed, and closes when the elapsed time counts to θ. The θ-window is intended for dual use – safeguarding against frequent actuation and ensuring minimal durability. The post-actuation context is durable if during θ-window, the context always satisfies the σ-margin specification, *i.e.*, $d_t \in [d-\sigma, d+\sigma]$, for any d_t in θ-window. When unspecified, the default values for the above option parameters are $\sigma = 0$ (no deviation allowed), $\tau = 0$ (only instant actuation allowed), and $\theta = 0$ (no requirement of durability).

For the purpose of simplicity, we use shorthand notations $W[e]$ or $W[e](d)$ to represent write operations in this document, indices and optional parameters σ, τ, and θ appear in our notations only when they are relevant to the discussion.

Pervasive Transaction Use Cases

As introduced in the previous section, the free operation and additional parameters δ, σ, τ, and θ are designed to provide enriched safety semantics to our pervasive transaction model. In this section, we use a few use case scenarios to showcase how our pervasive transaction model benefits from these designs, and empowers

programmers to compose safer and more practical applications in pervasive environment.

Scenario A - A Use Case for δ. The traffic law in China mandates that highway toll stations should release toll gates (free of toll charge) in conditions of heavy traffic to avoid congestions. A smart toll station system can be developed using a traffic sensor (such as inductive loops in roadbeds) to detect traffic volume (context entity e) and an automatic gate as an actuator. The gate opens to release traffic if the reading of e exceeds a pre-assigned threshold, and closes if otherwise. Because the traffic volume is a real time value that changes constantly, one undesirable but possible situation would be that the gate opens/closes frequently as a result of the fluctuation of e above and below the threshold value. Assigning a parameter δ to traffic volume sensing can address this problem, as it ensures the transaction would tolerate some fluctuation of traffic, resulting in calmer and smoother traffic controls.

Scenario B - A Use Case for σ. Conflicts arise as each service and application competes to derive the optimal context of its own. For example, two temperature control services (as agents representing two different users) may compete for AC control when both users are present in the same space. A conflict will arise, when their optimal temperature settings are different (despite the possibility that they may be close). Translating to our pervasive transaction model, this means at least one of the transactions will have to be aborted. Parameter σ allows for opportunities for resolving such conflict by providing reasonable relaxations for intended context. In this case, if one user's intended temperature is $65 \pm 3°\mathrm{F}$ (i.e., $d = 65, \sigma = 3$), and the other user's intended temperature is $70 \pm 2°\mathrm{F}$ (i.e., $d = 70, \sigma = 2$), the space will utilize the σs to

determine a mutually acceptable temperature of $68°\mathrm{F}$. This relaxation makes it possible for our transaction processor to use a single physical actuation to implement two concurrent logical writes (to temperature as a context entity). Therefore no transaction will be aborted in this case.

Scenario C - A Use Case for τ. Overcrowding is a safety concern in public spaces such as clubs, theaters, and stadiums. It is required by law that the population in such spaces should not exceed their designed capacities. To enforce these regulations, a smart space can utilize sensors (such as presence sensor and cameras) to monitor its population. In the event of an overcrowding situation, the space can lock down the entrances and prompts inhabitants to exit the space through visual or audio cues. However, this is no guarantee that these actions will achieve the desired effect (of decreased population) due to many factors. For example, visual or audio channels may not be effective in a busy environment, or inhabitants are simply unwilling to follow the prompt. Even if the persuasion channel is effective and inhabitants eventually follow the persuasion cue, the time delay between the actions and the derived effect will be not negligible. How long is the application willing to wait? Parameter τ enables a programmer to assign such time window to actuations to wait for delayed effects to materialize. In our case, if the population decreased below the capacity within the programmer-assigned τ-window, the actuation is considered successful; otherwise, it is considered failure and will cause the transaction to abort, which would typically require the space to enact stronger actions (such as alarms or calling law enforcement) to enforce the regulations.

Scenario D - Use Cases for θ. In Chapter 1, we described a scenario where a door actuator, represented by a software service, is subjected to over-frequent actuation (constant interchanging locking and unlocking) because the service fails to impose proper restrictions on its usage. Parameter θ provides the semantic power for such restriction. By allowing a transaction to specify a durability window (θ-window) for a door actuation, we will be able to prevent another actuation from immediately overriding its achieved context. (Because the other actuation has to wait for at least θ time before it acquires access to the door actuator.) θ-window is also useful when an application intends its achieved effect to last. For example, a smart house automatically locks its doors and windows at night for security. Such effects (locked doors and windows) are intended to last throughout the night, leading to a long-live transaction. By assigning a θ-window (θ in this case will be the length of the night) to the actuation, the space will be responsible to ensure the durability of such effects within θ-window. During this window, transactions attempting to unlock the doors and windows will be denied access. When external factors (for example, a human) override such effects, the space will notice a violation of the θ-constraint, abort the transaction, then enact a safety transaction to recover (by re-actuating the door and window locks), and report such violation to the users/admins (if the person who unlocked the door is unauthorized).

Scenario E - A Use Case for Free Operation. Falling is one of the most fatal hazards for elderly people in their homes, as well as for workers in construction sites. One technique to detect falling is to use 3-axis accelerometer. A sudden burst of vertical acceleration suggests a high possibility of falling, which calls for certain emergency actions to be taken (for example, contacting caregivers in the elderly home scenario,

and strengthening safety harness in the construction site scenario). Translating these scenarios to pervasive transactions, we would use a read operation to represent acceleration monitoring, followed by one or multiple write operations to represent the actions to be taken. However, one problem with this translation is that a pervasive transaction, by default, prohibits its view from changing after its read. In other words, if the accelerometer reading changes after the fall detection, the transaction would consider its view inconsistent and therefore aborts. Obviously in this case, “view consistency” is an unnecessary requirement – once a fall is detected, the emergency actions should be completed, regardless how the accelerometer reading changes afterwards. We address this problem by using the free operation. By assigning a free operation over the accelerometer reading, the pervasive transaction processing is no longer dependent on the view of accelerations.

The above scenarios showcase the semantic power of our pervasive transaction model. With the operation and parameters defined, application programmers are able to specify both safety requirements and relaxations that are practical in pervasive computing environments. To enforce these requirements and utilize the relaxations, runtime protocols/mechanisms are needed. In the following section, we introduce the Context Lock protocol, a locking protocol that regulates the runtime transaction processing by utilizing the semantic power in our pervasive transaction model.

Pervasive Transaction Programming Interface

Programmers formulate pervasive transactions using the transaction interface, in which, operations *tx_start()* and *tx_end()* are provided to declare the start and end of a transaction. Programmers are also provided with a list of all available operations and operands. Programmers can compose these operations between the *tx_start()* and

tx_end(), which will be assembled sequentially into a transaction at runtime according to the execution order of the program.

The Context Lock Protocol

The idea of locking context is two-pronged: (A) *isolation* - we hope to control the level of isolation among conflicting context operations (i.e. sensing and actuations) by issuing locks, and to provide a safer execution environment for concurrent pervasive transactions by separating conflicting transactions and limiting interference from the physical world; (B) *integrity* - we hope to maintain context within the prescribed context envelope by locking out risky operations on context entities. For each context entity in the space, if a lock can be assigned to protect its value from the impermissible range, then the entire pervasive space will conform to the overall integrity constraints, hence considered as safer. In this section, we introduce the concept of context lock, describe its elements and mechanism, and present a lock protocol that is proven to achieve the above two goals.

Pervasive Isolation Semantics

The issue of isolation has been well researched in conventional database systems. However, when translating this issue into the realm of pervasive computing, we notice a semantic change in the notion of isolation. Hence many well established database theories and approaches addressing the issue of isolation are no longer suitable. In this section, we first provide three observations of pervasive transactions to identify the semantics of isolation in pervasive computing.

First, there exist common patterns for pervasive transactions. Most pervasive transactions are issued by pervasive applications and services, which share typical

patterns when modeled by sensing (R) and actuation (W) operations. We observe four basic R-W patterns that constitute many pervasive transactions:

- *R*, which represents transactions with (one or many) sensing operations only.
- *W*, which represents transactions with (one or many) actuations operations only.
- *W-R*, which represents transactions that use sensing to verify the effects of completed actuations.
- *R-W*, which represents transactions that use sensor readings to determine the next actuations operations.

The above patterns represent four classes of pervasive applications with different requirements for isolation. For example, R-type transactions are typical for most monitoring and surveillance services. A sensor reading only reflects the context observed at the time of sensing. Sensing is typically performed at a pre-set pace. Previous senor readings do not affect the timing of the next sensing operations. R-type transactions do not require exclusive accesses to read the target context entities. In fact, they are used for context monitoring because other transactions and external factors may modify the context at any time. W-pattern represents a class of less common transactions, which perform actuations only. A W-type transaction could fail if a concurrent transaction also modifies the same context entities with an inconsistent target value. W-R pattern is common for applications that actively change the physical space through actuations (W), whose effects are then validated by sensor readings (R). Without isolating a W-R transaction, conflicting transactions and external events could interfere with the transaction, resulting in invalid effects of W. R-W pattern represents a class of transactions that react to context changes. These transactions make decisions to actuate based on sensor readings. Interferences are considered most harmful to this type of transaction - the decision of W could be based on an obsolete reading if the context entity is corrupted by interfering transactions and events between R and W. In

summary, we observe the three typical patterns for pervasive transactions require distinct levels of isolation. While some transactions are open to concurrent executions, others may suffer from conflicting transactions and interfering events, which cause inadequate or even wrongful executions.

Second, pervasive transactions need to embrace inaccuracies. Pervasive systems are inherently imprecise. Sensors produce readings within a certain accuracy and resolution. Actuators operate within limited precisions. Both sensing and actuations are performed in the physical space, a domain that is often dynamic, even volatile. Even if we have a pervasive system that perfectly isolates all conflicting transactions from each other, the interference from the physical world would still inevitably affect the result of these transactions. Moreover, many pervasive applications are mindful of these physical limitations, allowing room for inaccuracy in their executions. (For example, an electric oven service would accept temperature accuracy within 5 degrees.) Hence, to manipulate and protect context in pervasive spaces with the same precision as we manage data in database systems, is not only impractical but also unnecessary. This inaccurate nature of pervasive space requires us to shift focus on the issue of isolation for pervasive transactions. We argue that the goal here is not to establish an exact and uniform isolation for all transactions, but to provide customized isolations for transactions, each exposing only to its own acceptable degree of conflicts and interferences.

Third, conventional isolation levels are not suitable for pervasive transactions. Most database systems use popular isolation levels (such as UNCOMMITTED, COMMITTED, REPEATABLE, AND SERIALIZABLE) to control the behavior of

transactions. We find these well-established isolation levels inadequate for pervasive transactions. For example, consider “READ COMMITTED”, which aims to avoid “dirty read” - a transaction reads data that has been modified by another transaction that later rolls back. However, “dirty read” is hardly a concern for pervasive transactions, which never roll back by design. Unlike database transactions which rollback to discard tentative changes, any updates of context by pervasive are directly materialized into the physical world, where no changes are tentative. Another example is “REPEATABLE READ”, which prevents the phenomenon of “non-repeatable read” – any change of data that create inconsistent views between two reads. However, by this standard, most read in pervasive transactions are non-repeatable because sensor readings are subject to physical disturbance. A reasonable approach would be to relax the definition of “repeatable read”: a sensing operation is considered as repeatable if the change of the context does not exceed a tolerated threshold. Also, instead of applying a uniform level of isolation, we should allow individual transaction to specify its own degree of tolerance.

These observations lead us to rethink the semantics of isolation in the domain of pervasive computing, which are summarized as concept of “pervasive isolation”: (1) pervasive isolation is a property to protect the execution of a pervasive transaction from conflicts raised by concurrent transactions, as well as interference by external factors such as physical and human activities; (2) unlike database transaction which remains completely unaware of concurrent transactions during its execution, a pervasive transaction is mindful of potential conflicts and interference, and desires certain levels of isolation to limit the effects of these conflicts and interference; (3) individual pervasive

transaction can have distinct level of isolation depending on the purpose of the application.

To address the need for pervasive isolation, we introduce a variation of lock concept, namely, the context lock. We first describe two basic types of context lock – view lock and update lock, present their formal definitions, then introduce a lock management protocol for pervasive transactions.

Context Lock Types

In this section, we introduce two lock types: view lock and update lock.

A view lock isolates a transaction from other transactions and external events to ensure the quality of its reading. Based on the needs of different pervasive transaction patterns, we sketch three view lock types:

- $vl1$, which indicates the lock owner (a transaction) does not want the context entity to change while holding the lock. $vl1$ is the most restrictive among all read lock types by excluding any context changes.
- $vl2$, which indicates the lock owner allows any change to the context entity while holding the lock. $vl2$ places no restrictions on concurrent transactions or external events and is most suitable for R-type transactions (monitoring applications).
- $vl3$, which indicates the lock owner allows the context entity to change within a certain degree while holding the lock. As a relaxed from of $vl1$, $vl3$ is purposed for context entities with continuous domains where disturbance of sensor readings are practically inevitable. An example of such context entity would be temperature.

The above three view lock types cover the need for read protection by pervasive transactions in most scenarios. We present a formal definition of view lock in a generalized form below:

Definition 8-11. View Lock. A view lock, $vl[e](\delta)$, protects the view of context entity *e* to be consistent to the owner of the lock (transaction *T*). A view lock grants the

read access to the lock owner. The view of *e* is considered *consistent* if *e* satisfies the δ*-constraint* during the lifetime of view lock.

When a transaction is granted a view lock over context entity *e*, the lock ensures *e* to be "stable" by limiting the magnitude of change of e within a user-specified threshold *δ*. $vl1$ and $vl2$ are considered as special cases of the view lock where *δ* equals 0 and ∞ respectively. The view lock protects a context entity against incompatible concurrent transactions by denying write access.

As previously discussed in Chapter 7, the difference between pervasive systems and database are apparent despite of the similarities. As a result of these differences, the same strict isolation control in databases cannot be imposed on the pervasive transactions without proper relaxations. The parameters σ, τ, and θ defined previously provide several opportunities for relaxations on the writer side:

- First, the deviation relaxation threshold, σ, accommodates the possibility for concurrent write operations if there exist a mutually acceptable range of intended context for these writes.
- Second, the temporal relaxation window, τ, enables a large class of write operations that actuate through interactions with the physical environment or human. The completion of such write operation takes longer time than the actuation of the physical device alone, which requires longer isolation protection provided by context locks.
- Third, the durability relaxation window, θ, relaxes the requirement of permanent durability (which is rarely intended in pervasive spaces) to temporary durability in a timed window, which is achievable through monitoring after the completion of write. Supporting this feature requires augmentation of the semantics of update locks to utilize the same monitoring service used by view locks.

In this section, we present our update lock definition that implements the above relaxations.

Definition 8-12. Update Lock. A update lock, $wl[e](d, \sigma, \tau, \theta)$, is assigned over a write operation $W[e](d, \sigma, \tau, \theta)$ to ensure the following:

- σ-constraint: $W[e]$ shares concurrent access with another operation only if they are mutually compatible (as defined in Definition 8-11).
- τ-constraint: The post-actuation context d' is attained (within the σ-margin of deviation) before τ-window expires. Otherwise, the update lock will be revoked, the owner transaction aborted.
- θ-constraint: The post-actuation context d' conforms to the deviation margin requirement: *i.e.*, $d' \in [d - \sigma, d + \sigma]$ throughout θ-window. Otherwise, the update lock will be revoked, the owner transaction aborted.

For the purpose of simplicity, we use the shorthand notation $wl[\mathrm{e}]$ to represent an update lock in this document, parameters d, σ, τ, and θ appear in our notations only when they are relevant to the discussion.

An update lock provides the necessary isolation to ensure that the protected write operation achieves the target context and maintains its durability within the desired time windows. Once a transaction acquires an update lock, it has both read and write accesses to the locked context entity. As definition 7-10 suggests, update locks are not necessarily exclusive. A transaction may share its access to a context entity e with other transactions as long as their operations over e are mutually compatible. We define the compatibility of operations as follows.

Definition 8-13. Compatibility of Operations. The compatibility among all operation types are defined as following:

A free operation, $F[e]$, is compatible with any operation;

A read operation, $R[e](\delta)$, is compatible with any operation except a write operation whose projected post-actuation context could violate the δ-constraint, *i.e.*, a

write operation $W[e](d, \sigma)$ is compatible with $R[e](\delta)$ only if $[d - \sigma, d + \sigma] \subseteq [c - \delta, c + \delta]$, c is the current value of e.

Two write operations, $W_{ij}[e](d_1, \sigma_1)$ and $W_{ik}[e](d_2, \sigma_2)$, are compatible only if their σ-margins are mutually satisfied, *i.e.*, $d_1 \in [d_2 - \sigma_2, d_2 + \sigma_2]$ and $d_2 \in [d_1 - \sigma_1, d_1 + \sigma_1]$.

The full compatibility matrix is presented in the Table 8-2 as a summary of the above definition. Note that compatibility is determined entirely by the operation types and their parameters.

The role of the above compatibility matrix is to determine the compatibility among operations and their corresponding locks so that processing incompatible operations is disallowed. However, enforcing compatibility only partially fulfills AI^{2}D, leaving a greater role for the context lock protocol (next section) to ensure AI^{2}D.

The Protocol

The context lock protocol contains the following policies:

- Policy 1. A transaction T must obtain a view lock to read, and an update lock to write;
- Policy 2. To obtain an update lock, a transaction can only upgrade it from a view lock;
- Policy 3. A transaction T can release a view lock over e if there are no more operations in T to be processed that are dependent on the view of e (either as indicated explicitly by an $F[e]$ operation, or by reaching at the end of T), in other words, in order to perform an operation O, T must possess the view locks over the context entities which O is dependent on;
- Policy 4. A transaction T can degrade an update lock into a view lock after write, but cannot directly release an update lock;
- Policy 5. A transaction T succeeds to obtain a lock over e only if the requested lock is compatible with all locks in effect over e;
- Policy 6. If transaction T does not succeed to obtain a lock, it will be aborted.

For operation $O[e]$ over context entity e, we denote four different types of lock operations:

- $Ol[e]$: obtain a view lock;

- $Or[e]$: release a view lock;
- $Ou[e]$: upgrade a view lock into an update lock;
- $Od[e]$: degrade an update lock into a view lock.

Proof of Correctness

First, we present the formal definition of a few concepts to be used in our proof.

Definition 8-14. History. When a set of transactions execute concurrently, their operations may be interleaved. Also, these operations may interleave with external events. We model such execution by a structure called a history. Each operation $O[x]$ receives a timestamp: $TS(O[x])$, which marks the processing time of the operation. A history over a set of transactions $\{T_i\}$ stores timestamps of all operations in the transaction set, denoted as: $h = \{TS(O_{i,j}) \mid T_i \in \boldsymbol{T}, O_{i,j} \in T_i\}$, is a history over $\boldsymbol{T} = \{T_1, \dots, T_n\}$, where T_i is a transaction, and $O_{i,j}$ is an operation of T_i.

Definition 8-15. Conflict. In a history h, two operations $O_i[x]$ and $P_j[y]$ from two transactions T_i and T_j are in conflict if (1) they are incompatible, or (2) $P_j[y]$ is view-dependent on x if $TS(O_i[x]) < TS(P_j[y])$.

Definition 8-16. Conflict Equivalence of Histories. We define two histories h and h' to be equivalent (≡) iff.:

- they are defined over the same set of transactions and have the same operations,
- they order conflicting operations of committed transactions (and external events) in the same way.

Definition 8-17. Correctness of a history set. A history set H is correct if for each history $h \in H$, either $h \in H(2PL)$ or h is conflict-equivalent to a 2PL history, i.e., we can transform h into h' such that $h' \in H(2PL)$. And in such transformation, no pair of conflicting operations will be re-ordered.

Definition 8-18. Early Release. We define early releases over two types of context locks:

- Early release of a view lock: in history h, for any transaction T, if there exist operations $P[x]$ and $Q[y]$in T (x and y are context entities), such that $TS(Pr[x]) < TS(Ql[y])$ or $TS(Pr[x]) < TS(Qu[y])$, then there is an early release of a view lock.

- Early release of a update lock: in history h, for any transaction T, if there exist operations $P[x]$ and $Q[y]$in T, such that $TS(Pd[x]) < TS(Qu[y])$ or $TS(Pd[x]) < TS(Ql[y])$, then there is an early release of an update lock.

Now we start our proof of the context lock protocol.

Given a set of transactions, the class of histories generated by our context locking protocol is denoted as $H(CL)$. We divide this class into two subclasses:

$H(CL) = H_1 \cup H_2$, where:

- H_1 = {h_i | there exists at least one early release of some context lock in history h_i}
- H_2 = {h_i | there does not exist any early release of any context lock in history h_i}

To prove the correctness of $H(CL)$, we show correctness of H_1 and H_2 respectively.

Correctness of H_1

We consider two cases:

Case 1. There are only early releases of view locks in H_1. Given $h \in H_1$, by definition, we can find operations $P_i[x]$ and $Q_i[y]$ in T_i, such that $TS(Pr_i[x]) < TS(Ql_i[y])$ or $TS(Pr_i[x]) < TS(Qu_i[y])$.

Case 1.1. If there does not exist an operation $O_j[x]$ from another transaction T_j, such that $TS(Pr_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$), then the early release of $Pr_i[x]$ does not affect other transactions. We can swap $Pr[x]$ and $Ql[y]$ (or $Qu[y]$) without changing the order of any operations in history h. We can keep swapping such

early release pairs of lock operations until there are no more early releases in the history. The resulting history $h' \in H(2PL)$.

Case 1.2. If there exists an operation $O_j[x]$ from another transaction T_j, that $TS(Pr_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$), we can prove that $O_j[x]$ does not affect $Q_i[y]$. (Proof: according to context locking protocol (Policy 3), $Q_i[y]$ is not dependent on the view of x. Because $O_j[x]$ is an operation over x, it could only affect the view of x. Therefore $O_j[x]$ does not affect $Q_i[y]$.) The processing of $Q_i[y]$ is oblivious to $O_j[x]$, *i.e.*, as if $O_j[x]$ does not exist, which is same as in Case 1.1.

Case 2. There are early releases of update locks in H_1. Given $h \in H_1$, by definition, we can find operations $P_i[x]$ and $Q_i[y]$in T_i , such that $TS(Pd_i[x]) < TS(Ql_i[y])$ or $TS(Pd_i[x]) < TS(Qu_i[y])$.

Case 2.1. There does not exist a lock release operation $Pr_i[x]$ such that $TS(Pd_i[x]) < TS(Pr_i[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$).

Case 2.1.1. There does not exist an operation $O_j[x]$ from another transaction T_j, such that $TS(Pd_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$). Similar to Case 1.1, we can transform h into h' such that $h' \in H(2PL)$, without changing ordering of any operations in h.

Case 2.1.2. There exists an operation $O_j[x]$ from another transaction T_j, such that $TS(Pd_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$). We can show $O_j[x]$ does not affect $Ql_i[y]$: operation $Q_i[y]$ is either dependent or independent on the view of x. If independent, then the processing of $Q_i[y]$ is oblivious to $O_j[x]$, similar to Case 1.2; if dependent, we can show $O_j[x]$ does not affect the view of x: since there is no lock

release $Pr_i[x]$ such that $TS(Pd_i[x]) < TS(Pr_i[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$), T_i must possess a view lock over x when $O_j[x]$ is processed, according to the locking protocol $O_j[x]$ does not affect the view of x (otherwise a lock will not granted to $O_j[x]$). Because $O_j[x]$ does not affect the view of x, $O_j[x]$ does not affect $Q_i[y]$.

Case 2.2. There exists a lock release operation $Pr_i[x]$ such that $TS(Pd_i[x]) < TS(Pr_i[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$).

Case 2.2.1. There does not exist an operation $O_j[x]$ from another transaction T_j, such that $TS(Pd_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$). Similar to Case 1.1, we can transform h into h' such that $h' \in H(2PL)$, without changing ordering of any operations in h.

Case 2.1.2. There exists an operation $O_j[x]$ from another transaction T_j, such that $TS(Pd_i[x]) < TS(O_j[x]) < TS(Pr_i[x])$. We can show $O_j[x]$ does not affect $Q_i[y]$: T_i is possessing a view lock over x when $O_j[x]$ is processed, according to the locking protocol, $O_j[x]$ does not affect the view of x (otherwise a lock will not granted to $O_j[x]$). Because $O_j[x]$ does not affect the view of x, $O_j[x]$ does not affect $Q_i[y]$.

Case 2.1.3. There exists an operation $Pr_i[x]$ such that $TS(Pr_i[x]) < TS(O_j[x]) < TS(Ql_i[y])$ (or $TS(Qu_i[y])$). Then $O_j[x]$ does not affect $Q_i[y]$, see Case 1.2.

Correctness of H_2

We prove $H_2 \subseteq H(2PL)$ by contradiction. Suppose there exists a history $h \in H_2$ such that $h \notin H(2PL)$. Then there exist operations $P[x]$ and $Q[y]$ such that one of the following is satisfied:

$TS(Pr[x]) < TS(Ql[y])$ or,

$TS(Pr[x]) < TS(Qu[y])$ or,

$TS(Pd[x]) < TS(Qu[y])$ or,

$TS(Pd[x]) < TS(Ql[y])$.

In either case, by definition, we have an early release in h. Therefore $h \notin H_2$, a contradiction.

Pervasive Transaction Management Architecture

Architecture Overview

In this section, we present the pervasive transaction management architecture, which provides a reference design to support our proposed pervasive transaction model and the context lock protocol. The overview of the architecture is presented in Figure 8-4. The core of the transaction management architecture is highlighted in grey, which includes the following five modules:

- *Transaction Pool*: a structure that manages and stores the pervasive transactions submitted by the applications;
- *Transaction Processor*: the processing engine that oversees the lifetime of each transaction by requesting/releasing context locks, executing operations, and committing/ aborting a transaction.
- *Context Lock Manager*: the management component that responds to all context lock requests submitted by the transaction processor;
- *Context Map*: a structure that maintains all the context entities - the operands of pervasive transactions;
- *Context Monitor*: a component that provides monitoring services over context entities to detect violations of context constraints subscribed by context locks.

In the remainder of this section, we present the design of each module in detail and introduce the interfaces for inter-module interactions. We will also present the key algorithms designs for selected modules. A full data structure design of the pervasive transaction processor written in Java is presented in Appendix D.

Transaction Pool

Function. The transaction pool provides a structure that stores all the pervasive transactions submitted by applications that are ready for or undergoing processing.

Applications can submit pervasive transaction requests to the transaction pool. Once a transaction is admitted into the pool, a pervasive transaction structure (*instance*) is created, which opens spaces for storing critical transaction information such as operation sequence and the runtime status. The transaction scheduler determines the order of processing among these transactions by employing multiple alternative scheduling strategies such as *FIFO*, *Round-Robin* and *SJF*.

Components*.* Key components of the transaction pool including the following:

- Transaction Pool Manager, which manages the life cycle of pervasive transactions. Applications can submit/update a transaction request to the transaction pool through the transaction pool manager. Once a transaction is admitted into the pool, the pool manager will monitor its runtime status. It will notify the corresponding application on the commit or abort of a transaction, and exit the transaction from the pool. The transaction pool manager is also responsible for validating transaction requests by checking the parameters and sequences of the operations issued. For example, it will invalidate the request to issue a read/write operation over *e* if *e* is previously freed by the same transaction.

- Transaction Scheduler, which determines the order among the transactions in the pool for the Transaction Processer. Multiple scheduling strategies could be implemented in the scheduler. As a baseline implementation, we propose to use the Round-Robin algorithm to maximize the concurrency among transactions.

- Pervasive Transaction Instance, which assigns memory spaces that store transaction information needed for processing. In each transaction instance, the operation sequence for the transaction (read, write and free) is stored. For each transaction, it also stores the *submission status* (e.g. whether an application has started a transaction by *tx_start()*, or closed the transaction by *tx_end()*), and *processing status* (e.g. the index of the next operation in the sequence to be processed).

Interface. The transaction pool provides the following two interfaces:

- Interface toward applications. Applications can utilize the pervasive transaction interface (see Chapter I.) to customize, assemble and submit transactions to the transaction pool manager. At runtime, an application requests the transaction pool to assign a new transaction instance through tx_start(). The application can continue to update the transaction by appending (read, write or free) operations to the instance, until it closes the transaction with tx_end(), which means the operation sequence of the transaction instance is finalized.

- Interface toward transaction processor. The transaction processor retrieves transaction instances from the pool one at a time through the getNextTransaction() interface provided by Transaction Scheduler. The content of the transaction instance is completely visible to the transaction processor. Once the processor retrieves an instance, it checks to runtime status to find the first operation in the sequence that has not been processed, then requests the necessary locks needed to perform the operation. If the locks are granted, it executes the operation and updates the runtime status of transaction. Then it continues to retrieve the next transaction to be processed from the scheduler.

Key Structures/Algorithms. An important aspect of the transaction pool design is the data structure that stores all the pervasive transaction instances in the pool. While there are many options available, the optimal structure is largely dependent on the scheduling strategy to be employed. As a baseline implementation, we propose to use a linked list as the container of all transaction instances.

Transaction Processor

Function. The transaction processor is the key module of the entire pervasive transaction management architecture. It is responsible for issuing lock requests, processing read/write operations, and committing or aborting transactions. The transaction processor handles one operation at a time – within what we call a transaction cycle. At the start of the cycle, the processor retrieves the transaction to be processed for the scheduler. It identifies the next operation to be executed in the transaction. It then looks up the lock table in the context map to determine if any lock (obtain/upgrade) action is need to perform the operation. If that is the case, it issues a lock request to the context lock manager. Once the lock request is granted, it process the operation by performing read or writes to the corresponding context entity stored in the context map. After the operation is successfully processed, it enacts the lock

scheduler again to see if any lock degrade/release action can be performed. Finally it updates the runtime status in the transaction instance and finishes a transaction cycle.

Components. The transaction processor consists of the following key components:

- Lock Scheduler, which determines and performs lock request/release actions pre/post operation processing.
- Operation Processor, which performs read/write operations over context entities stored in the context map.
- Abort Manager, which identifies situations where abort is needed and perform the fail-safe abort.
- Commit Manager, which performs the transaction commit.

Interface. The transaction pool provides the following two interfaces:

- Interface toward Context Lock Manager. The transaction processor interacts with the context lock manager through the lock scheduler by exchanging lock request/response messages.
- Interface toward Context Map. The transaction processor processes transactions operations over context entities, which are stored in the context map. We will present the structure of the context map and explain the interactions between the context map and the transaction processor in the following section.

Context Lock Manager

Function. The context lock manager responds to lock requests from the transaction processor. It enforces the context lock protocol by ensuring that (1) co-existing context locks are always compatible with each other (through Compatibility Tester); and (2) when a lock constraint (such as δ-constraint and σ-constraint) is violated, its corresponding lock will be invalidated. In addition, the context lock manager also detects and handles deadlocks.

Components. Key components of the context lock manager include the following:

- Deadlock Detector, which detects the deadlock situations by analyzing the wait-for graph for transactions.
- Compatibility Tester, which checks if a requested lock is compatible with all context locks in effect.

Interface. The context lock manager is dependent on the context monitor module, which provides monitoring services over δ-constraints and σ-constraints. When the monitor detects a violation of the constraint, it notifies the context lock manager by invoking onViolation() interface. The context lock manager reacts to such violation by revoking the corresponding context lock, which potentially results in a transaction abort in the transaction processor.

Context Map

Function. The context map stores all the operands of pervasive transactions – context entities. For each context entity, a structure (instance) is created to store information related to the context entity.

Safety Transaction Mechanism

Safety transactions are a class of transactions initiated by the pervasive system to maneuver context within the context envelope by issuing sensing and actuation operations. The goal of safety transactions is to enable a more intelligent and autonomous pervasive system in face of impermissible and risky situations.

When context is escaping from the desired envelope, safety transaction is expected to enact proper sequences of actuations to counter the erroneous context, restore and stabilizes the desired context. Safety transaction is a best-effort mechanism. Limited by the sensing and actuation capabilities of the system, safety transaction does not guarantee the elimination of all safety risks, but only provides additional protection during risky scenarios.

In this section, we first present the definitions of safety transaction and its elements. Then we describe an algorithm to safety transaction composition.

In Chapter 5, we defined that sensing and actuation operations in the above definitions are *logical* operations. The definitions describe the logical effects of these operations within the context space. While the link between physical sensors and logical sensing operation is obvious, the link between logical actuation into the physical domain, however, is not necessarily trivial. Take a logical actuation $w(e, d)$ for example, there maybe none, exactly one, or multiple actuator devices available to perform $w(e, d)$ in the physical space. In addition, by performing operation $w(e, d)$, an actuator device possibly creates "side effects" - affecting context entities other than e, in a way may or may not known to the system. Therefore, translation between logical actuations and physical operations requires a proper actuator model to be established first [**Def. 1**].

Revisiting Actuators

1.1 *Actuator*. The pervasive system has a set of available actuator devices $\mathbf{A} = \{a_i\}$. An actuator a is defined by a parameter space $\boldsymbol{P}^a = \{p \mid p$ is an acceptable parameter combination for actuator $a\}$.

1.2 *Trace of Actuation*. To better understand the effect of actuations, a pervasive system keeps a record of the contextual traces for each executed actuation. The trace of an actuation w, is defined as a six-tuple: $trace_w = (a, p, c_{start}, c_{end}, t_{start}, t_{end})$, where a is the actuator device that performs the actuation, p is the parameter combination assigned to the device, c_{start} and c_{end} are the context at the start and end of execution respectively, and t_{start} and t_{end} are timestamps of c_{start} and c_{end} respectively. We show how t_{end} is determined in the context lock protocol.

Problem Formulation

Safety transaction has the same formal structure as any pervasive transaction. It consists of a number of sensing and actuation operations; and it is managed by the same transaction-processing engine. Safety transactions receive higher priorities in lock acquisition. A context lock granted a pervasive transaction will revoked if it is requested by a safety transaction.

One question remaining, however, is that how to find the suitable and most effective sequences of operations to fulfill the goal of risk mitigation. We formulate the problem of safety transaction composition into two sub-problems.

Problem A. *Composing safety transaction with logical operations*. For context space $\boldsymbol{C^P}$, given context envelope $\boldsymbol{\varepsilon}$ and current context c, compose a safety transaction $ST = \{\{w_i, r_j\}, <\}$ that pulls current context c into the envelope, i.e. find ST: $c \rightarrow \boldsymbol{c'}$, such that $\boldsymbol{c'} \in \boldsymbol{\varepsilon}$.

Problem B. *Translating from logical to physical operations*. Given a logical actuation $w(e, d)$, current context c, and actuation trace set $\mathbf{TR} = \{trace_i\}$, find the most appropriate actuator $a \in \mathbf{A}$ with parameter $p \in \boldsymbol{P^a}$, that changes the value of e to d.

In the next two sub-sections, we propose two algorithms to solve each of the above sub-problems.

Algorithm A. Composing safety transaction using logical operations

We present a procedure for problem A in pseudo-code in Figure 8-5.

To analyze the algorithm, we describe two technical details in Procedure A.

- *Distance function*. A natural measure of distance between two context points would be their “geometric distance” within the context space. Since context space uses context entity values, i.e. sensor readings as scalars, this geometric distance is not necessarily a good metric to measure the effort of context transformations,

which are accomplished by actuators instead of sensors. For example, two context points may have a close sensing-distance but a long actuation-distance because the actuator takes very long time to achieve such context transform. This observation inspires us to improve the algorithm by using actuation-based distance functions.

- *Pureness of actuations*. The success of the above algorithm relies on one assumption: for any actuation *w*(*e, d*) contained in the result *ST*, an actuator always exists to perform *w*(*e, d*), and the effect of such actuation is "pure" – it will not affect any context entities other than *e*. This assumption is often not realistic. To fix this problem, we propose a technique to minimize the "side effects" in our actuator selection discussed in problem B, and improve procedure A by using multiple loops to approximate target envelope, if a direct actuation is not possible.

Algorithm B. Translating from logical to logical operations

We present a procedure for problem B in pseudo-code in Figure 8-6.

Algorithm B relies on the observation that most actuations produces *repeatable* effects – if an actuation causes a change over a certain context entity, repeating the same actuation is likely to produce the same context change. This inspires us to delve into the history of actuation traces and utilize existing knowledge to find the appropriate actuations.

To limit the side effects of a physical actuation, we use cross products in the process of actuation selection, denoted as Φ. Smaller Φ indicates lower side effects over other context entities.

We use execution time Δt as the cost of actuation for the final actuation selection. The lower Δt, the faster the actuation achieves the desired context, the more efficient the safety transaction is.

Utilizing the previously discussed heuristics, we present the procedure for overall safety transaction composition in Figure 8-7.

Limitations of Safety Transactions

The safety transaction mechanism is most effective in pervasive spaces with "symmetric" capabilities of sensing and actuation. On one hand, the space relies on actuators to correct faulty context observable to its sensors; on the other hand, sensors are responsible to monitor the effect of actuations and ensure the safety and correctness of their executions. We say a pair of sensor and actuator is symmetric if they access and modify a shared set of context entities. Clearly, the more symmetric sensor-actuator pairs there are in the pervasive space, the more powerful the safety transaction mechanism is. However, existing pervasive systems typically have greater sensing power than actuation, which limits the possibilities for safety transaction composition.

In addition, safety transaction is a "best-effort" mechanism. Pervasive systems, utilizing existing knowledge of sensor and actuator operations, can only suggest a safety transaction that is mostly like to work. However, there is no guarantee of the effectiveness of safety transaction, because the affects of some actuations may not be repeatable. In critical and severe safety conditions, specific human knowledge and human intervention are more authoritative and effective than system's learned knowledge. The best application for safety transaction therefore is for maintaining safe and desired context, rather than averting severe and extreme context.

Table 8-1. State transition rules of pervasive transaction processing

State Transition	Condition / Action Pair	Description
init → *soa*	PT / -	The initialized transaction is treated as a pervasive transaction (PT).
soa → *exec*	LCK_REQ / LCK_GRT	A pervasive transaction proceeds to exec state if the requested locks are granted.
soa → *fsa*	LCK_REQ / LCK_REJ	A transaction proceeds to fsa state if a lock request is rejected.
soa →*pcmt*	TR_CMPLT / -	If all operations of a transaction have completed execution, the transaction proceeds to pcmt state.
exec → *soa*	EXEC_S / LCK_RLS	A transaction releases the context locks and returns to soa state if the execution of the current operation succeeds.
exec → *fsa*	EXEC_F / TR_ABRT	A transaction proceeds to fsa state if the execution of the current operation fails.
	LCK_RVK / TR_ABRT	A transaction is forced to fsa state if its possessed context lock has been revoked.
pcmt→*cmt*	WAIT<TO / TR_CMT	A transaction proceeds to cmt state if the context enters the target context envelope before time window TO closes.
pcmt→ *fsa*	WAIT≥TO/ TR_ABRT	A transaction proceeds to fsa state if the context fails to enter the target context envelope before time window TO closes.

Table 8-2. Compatibility Matrix of Operations.

	$F[e]$	$R[e](\delta_2)$	$W[e](d_2, \sigma_2)$
$F[e]$	Compatible	Compatible	Compatible
$R[e](\delta_1)$		Compatible	Compatible if $W[e]$ conforms to δ_1-constraint; otherwise incompatible
$W[e](d_1, \sigma_1)$			Compatible both σ_1 and σ_2 margins are satisfied; otherwise incompatible

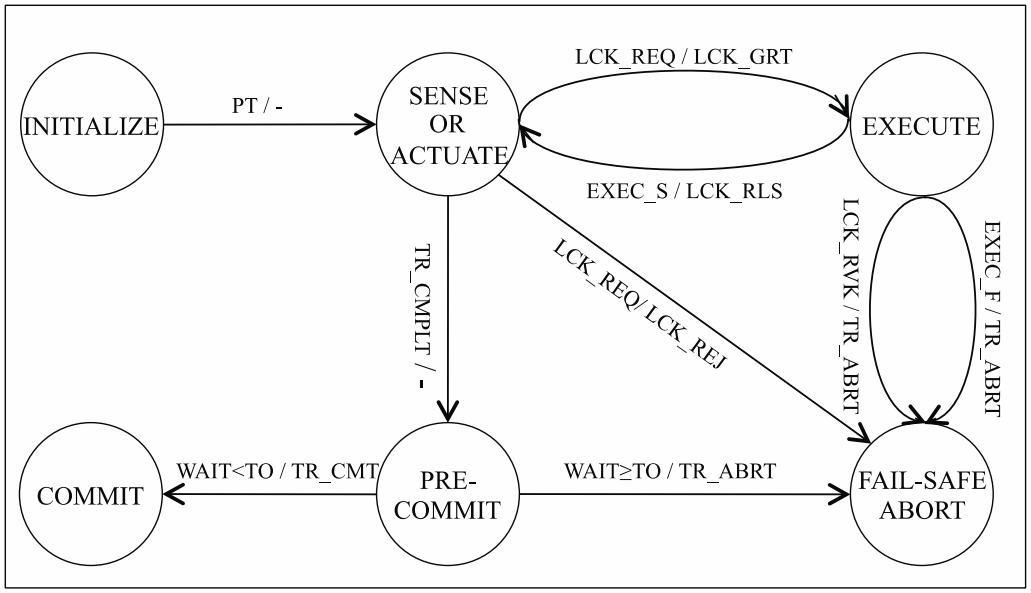


Figure 8-1. The state diagram of a pervasive transaction.

```
Procedure PRE_CMT (Ti)
    1.  condition = WAIT(TO);
    2.  if condition = "WAIT<TO"
    3.      TR_CMT(Ti)
    4.  else
    5.      TR_ABRT(Ti, "WAIT=TO")
End

Procedure WAIT (TO)
    1.  start timer;
    2.  while (t < TO);
    3.      for each e in E^P
    4.          execute r(e)
    5.      update current context: c = (r(e1), r(e2), ... r(en))
    6.      if (c ∈ εi)        // εi is the target context envelope of transaction Ti
    7.          break
    8.  if (t < TO)
    9.      return "WAIT<TO"
    10. else
    11.     return "WAIT=TO"
End
```

Figure 8-2. Procedure of pre-commit.

```
Procedure TR_ABRT (Ti, condition)
    1.  switch condition
    2.  case condition = "EXEC_F" || "LCK_RVK"
    3.      goto step 8;
    4.  case condition = "LCK_REJ"
    5.      goto step 15;
    6.  case condition = "WAIT=TO"
    7.      goto step 17;
    8.  for each op in Ti.operations AND op is in execution,
    9.      stop operation op by sending a STOP signal to the device (if possible);
    10. for each e in E^P
    11.     execute r(e)
    12. update current context: c = (r(e1), r(e2), ... r(en));
    12. if c ∉ ε        // εT is the overall system-wide context envelope
    13.     invoke ST: c→ε;
    14. for each L in L
    15.     LCK_RLS (L, Ti);
    16. update global context envelope ε = ε - εi//εi is the target envelope for transaction Ti;
    17. Ti.state = "FAIL_SAFE_ABORT";
    18. Q.dequeue (Ti);
End
```

Figure 8-3. Procedure of fail-safe abort (*TR_ABORT*).

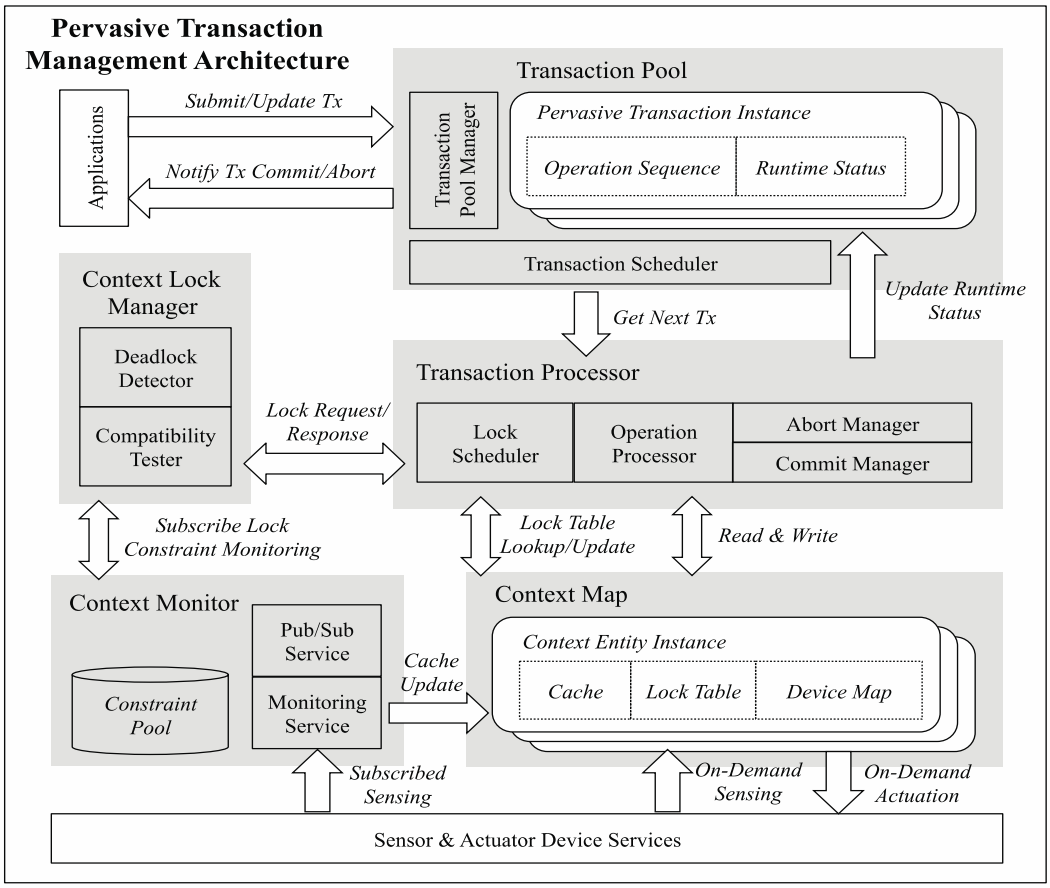


Figure 8-4. The Pervasive Transaction Management Architecture

Procedure A

1. Initialize $ST = \emptyset$;
2. For each context entity $e \in \boldsymbol{E}^{\boldsymbol{P}}$,
3. project c on dimension e, the projection denoted as $c|_e$;
4. project $\boldsymbol{\varepsilon}$ on dimension e , the projection denoted as $\boldsymbol{\varepsilon}|_e$;
5. if $c|_e \in \boldsymbol{\varepsilon}|_e$, go to step 2;
6. find $d \in \boldsymbol{\varepsilon}|_e$, such that the distance function $dist(c|_e, d)$ returns the minimal;
7. append $w(e, d)$ to ST;

Return ST.

Figure 8-5. Algorithm for composing safety transaction with logical operations.

Procedure B

1. query **TR** based on start and end context, and store result traces in **H**: $\mathbf{H} = \{trace_i \mid c_{start}|_e = c|_e, c_{end}|_e = d\} \subseteq \mathbf{TR}$, $c|_e$ denotes the projection of c on e;
2. if $\mathbf{H} = \emptyset$, go to step 8;
3. for each $trace \in \mathbf{H}$,
4. compute cross product $\Phi = \langle c_{start}, c_{end} \rangle \times \underline{e} \,/\, |\langle c_{start}, c_{end} \rangle|$, $\underline{e}$ is the unit vector of domain e;
5. if $\Phi > \eta$, remove the trace from **H**, η is a pre-defined threshold;
6. for each $trace \in \mathbf{H}$, compute $\Delta t = t_{end} - t_{start}$;
7. find $trace_i$ with the lowest Δt_i;
8. return (a, p) of $trace_i$.
9. return MESSAGE (no actuator available).

End

Figure 8-6. Algorithm for translating logical to physical operations.

```
Procedure SAFETY_TR_COMPOSE
    1.  Initialize ST = Ø;
    2.  for each context entity e ∈ E^P, append r(e) to ST;
    3.  execute ST to derive current context c;
    4.  initial_distance = min{dist(c, ε)};
    5.  call Procedure A;
    6.  for each w in ST
    7.          call Procedure B;
    8.          if Procedure B returns error, abort ST;
    9.          else execute actuation on actuator a with parameter p;
    10. for each context entity e ∈ E^P, append r(e) to ST;
    11. execute ST to derive current context c;
    12. distance = min{dist(c, ε)};
    13. if distance = 0, commit ST;
    14. else if distance < initial_distance, go to step 5;
    15. else abort ST;
    16. end.
End
```

Figure 8-7. Improved algorithm combining Procedure A and B.

CHAPTER 9
PERFORMANCE EVALUATION

Approach

In this chapter we examine and quantify the performance of the context lock protocol and several of the associated pervasive transaction concepts. To this end, an experimental approach based on event-driven simulation and benchmarking has been attempted. However, designing a pervasive transaction / smart space benchmark proved to be quite challenging. This is mainly due to the lack of data describing pervasive transactions – a novel concept that is yet to be established. Consecutively, synthesizing a benchmark for the simulator was an extremely challenging task.

In response, we adopted an alternative analytic approach to performance modeling and evaluation. The analytic models allowed us to zoom in on the critical aspects and conditions of a smart space running pervasive transactions. Specifically, we zoomed in on the probabilistic aspects of contention as well as relaxation opportunities (e.g., Free operation, δ, σ, τ, and θ).

We present two analyses in this chapter. In Analysis 1, we examine the effects and inter-relationships of the Free, δ and σ operations, under invariant conditions of τ and θ. In Analysis 2, we examine the opposite scenario – the effects and inter-relationships of τ and θ, under invariant conditions for Free, δ and σ. The division of the analysis into two parts was necessary to reduce the dimensionality of the problem.

Analysis 1 – Free, δ and σ Analysis

Objective

Analyzing the effect of {Free, δ, σ} on the overall pervasive transaction processing under the context lock protocol. The main performance metric we sought is the

probability to abort - $\mathrm{P_{abt}}$. A meritorious effect of any of the {Free, δ, σ} is one that can be measured as a decline in $\mathrm{P_{abt}}$ and vice versa.

We follow the notation scheme introduced in [45].

Assumptions:

- Single transaction class. All transactions can be characterized by the same profile (including the mean length of transaction, probability distribution of access pattern, write (or read) ratio, and free, δ and σ characteristics).
- No external events. We only consider aborts induced by incompatible lock requests (contentions) in Analysis 1.
- Transaction arrival rate λ. The combined arrival of new and restarted (due to abort) transactions is characterized by λ.
- Computation time overhead and lock acquisition/release time overhead are negligible. We only consider the processing time, i.e., time to read (sense) and time to write (actuate).

Transaction Model

The following are parameters that characterize pervasive transactions in this analysis:

- transaction arrival rate λ,
- access matrix A is a $I \times M$ matrix, where I is the total number of context entities in the space, and M is the mean length of pervasive transactions (i.e., number of read and write operations, excluding free, in a transaction). $A_{i,k}$ represents the probability that the kth operation of the transaction accesses (through read or write) the ith context entity.
- W(*resp*. $1 - W$) is the probability that an operation of the transaction (again, excluding *Free*) is a write (*resp*. read).
- mean distance to free F, is a parameter designed to characterize the occurrence of Free operations. F is defined by the average number of operations between an (read or write) access over a context entity and the free of the same context entity (either explicitly by a free operation or inexplicitly by reaching the end of transaction.
- mean time to read: R_{read} and mean time to write R_{write} (θ *and* τ window included), clarify what distribution we are assuming.

Lock Holding Time

The lock holding time for the access to i at the kth operation of a give transaction starts from processing the kth operation and ends when the lock is released. Because

the lock protocol prohibits transaction waiting for a lock, and we omit the lock acquisition and release overhead (insignificant compared to operation execution time), the lock holding time is essentially the total operation execution time between the kth operation and the lock release:

$$D_k = \sum_{j=k}^{\min(k+F,M)} [(1 - W_k)\, R_{read} + W_k R_{write}], \tag{9-1}$$

where W_k is probability that the kth operation of the transaction is a write. Using a universal write probability W, D_k can be further reduced to:

$$D_k = (\min(k + F, M) - k + 1) R_{op}, \tag{9-2}$$

where R_{op} is the mean processing time per operation and $R_{op} = (1 - W) R_{read} + W R_{write}$.

We know that the probability to access i at the kth step is $A_{i,k}$. Hence, the average lock holding time for context entity i per access is

$$Th_i = \frac{\sum_{k=1}^{M} A_{i,k} D_k}{\sum_{k=1}^{M} A_{i,k}} \tag{9-3}$$

Probability of Read or Write

The arrival rate of read accesses towards i is

$$\lambda_{read,i} = \lambda \sum_{k=1}^{M} A_{i,k} (1 - W), \tag{9-4}$$

while for write access, we have

$$\lambda_{write,i} = \lambda \sum_{k=1}^{M} A_{i,k}\, W. \tag{9-5}$$

The probability that an incoming access request towards i is a read and write request, respectively, is

$$P_{read,i} = \frac{\lambda_{read,i}}{\lambda_{read,i} + \lambda_{write,i}} \text{ and } P_{write,i} = \frac{\lambda_{write,i}}{\lambda_{read,i} + \lambda_{write,i}} \tag{9-6}$$

Probability of Contention.

When a read access occurs towards i, a contention is raised if i is locked by at least one incompatible update lock.

The probability that there exists one update lock is

$$P_{update,i} = \lambda_{write,i} Th_i \tag{9-7}$$

The probability that an update lock and a read access conflicts, denoted as P_{rw}, is dependent on the distribution of d, σ and δ.

When a write access occurs towards i, a contention is raised if i is locked by at least one incompatible update or view lock.

The probability that there exists one update lock is derived in Equation 9-7.

The probability that an update lock and a write access conflicts, denoted as P_{ww}, is dependent on the distribution of d, σ.

The probability that there exists one view lock is

$$P_{view,i} = \lambda_{read,i} Th_i \tag{9-8}$$

The probability that a view lock and a write access conflicts, denoted as P_{wr}, is the same as P_{rw}, i.e., $P_{wr} = P_{rw}$.

The contention probabilities for a read and write access towards i, respectively, are

$$\begin{gathered} PW_{read,i} = P_{rw} P_{update,i}, \text{ and} \\ PW_{write,i} = P_{ww} P_{update,i} + P_{rw} P_{view,i} \end{gathered} \tag{9-9}$$

Uniform-based Derivation of P_{rw} and P_{ww}

P_{ww}: Assuming 1. each context entity i has a continuous domain normalized to [0,1]; 2. the distribution of target value of write d in such domain is uniform; and 3. σ is

the normalized percentage, and is a constant for all writes (for simplicity). We can evaluate P_{ww} as the probability that two random variables, d_1 and d_2, do not satisfy: $d_2 - \sigma < d_1 < d_2 + \sigma$ and $d_2 - \sigma < d_1 < d_2 + \sigma$, *i.e.,*

$$\begin{aligned} P_{ww} &= 1 - P(d_2 - \sigma < d_1 < d_2 + \sigma\,)P(d_1 - \sigma < d_2 < d_1 + \sigma) \\ &= 1 - (1 - (1 - \sigma)^2)^2 \end{aligned} \tag{9-10}$$

P_{rw}: The probability of a read and write contention can be evaluated by the probability that the σ-constratin is violated (i.e., given the target value of write d and the current of value at the lock request c, the condition that $[d - \sigma, d + \sigma] \subseteq [c - \delta, c + \delta]$ is not true).

$$\begin{aligned} P_{rw} &= 1 - P(c + \delta > d + \sigma \text{ and } c - \delta < d - \sigma) = 1 - (1 - (1 - \delta + \sigma)^2) \\ &= (1 - \delta + \sigma)^2 \end{aligned} \tag{9-11}$$

The above derivation of P_{ww} and P_{rw} is based on the assumptions that distributions of both d and c are uniform. However, in certain scenarios, we find Gaussian distributions may be more realistic. For example, when a context entity i represents a physical aspect (such as temperature or humidity), it is often observed that its values can be fitted by Gaussian distributions. In the following section, we show an alternative derivation of P_{ww} and P_{rw} based on Gaussian distribution.

Gaussian-based Derivation of P_{rw} and P_{ww}

In order to apply Gaussian distribution, we normalize the domain of each context entity i into an infinite interval $(-\infty, +\infty)$. We assume two Gaussian distributions on random variables d and c, denoted by $d \sim \mathcal{N}(\mu_d, \sigma_d^2)$ and $c \sim \mathcal{N}(\mu_c, \sigma_c^2)$, respectively.

We derive P_{rw} and P_{ww} as follows:

P_{ww}: We know that $(d_2 - d_1) \sim \mathcal{N}(\mu_d - \mu_d, \sigma_d^2 + \sigma_d^2) = \mathcal{N}(0, 2\sigma_d^2)$.

$$P_{ww} = 1 - P(|d_2 - d_1| < \sigma\ )^2 = 1 - \Big(\mathrm{CDF}(\sigma; 0{,}2\sigma_d^2) - \mathrm{CDF}(-\sigma; 0{,}2\sigma_d^2)\Big)^2 = 1 - \left(\Phi\left(\frac{\sigma}{\sqrt{2}\sigma_d}\right) - \Phi\left(\frac{-\sigma}{\sqrt{2}\sigma_d}\right)\right)^2 = 1 - \left(2\Phi\left(\frac{\sigma}{\sqrt{2}\sigma_d}\right)\right)^2, \tag{9-12}$$

where $\Phi(x)$ is the cumulative distribution function (CDF) of the standard normal distribution, $\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}\mathrm{d}t$.

P_{rw}: We know that $(d - c) \sim \mathcal{N}(\mu_d - \mu_c, \sigma_d^2 + \sigma_c^2) = \mathcal{N}(\mu_d - \mu_c, \sigma_d^2 + \sigma_c^2)$.

$$\begin{aligned} P_{rw} = 1 - P(\sigma - \delta < d - c\ < \delta - \sigma) \\ &= 1 \\ &- [\mathrm{CDF}(\delta - \sigma; \mu_d - \mu_c, \sigma_d^2 + \sigma_c^2) \\ &- \mathrm{CDF}(\sigma - \delta; \mu_d - \mu_c, \sigma_d^2 + \sigma_c^2)] \\ &= 1 - \Phi\left(\frac{\delta - \sigma - \mu_d + \mu_c}{\sqrt{\sigma_d^2 + \sigma_c^2}}\right) + \Phi\left(\frac{\sigma - \delta - \mu_d + \mu_c}{\sqrt{\sigma_d^2 + \sigma_c^2}}\right) \end{aligned} \tag{9-13}$$

Probability to Abort.

For the kth operation in a transaction, the probability that such operation completes successfully (i.e. free of contention) is

$$PS_k = \sum_{i=1}^{I} \{A_{i,k}\,[P_{read,i}\big(1 - PW_{read,i}\big) + P_{write,i}\big(1 - PW_{write,i}\big)]\} \tag{9-14}$$

Probability for a transaction to commit, P_{cmt}, is the probability of contention-free access for all its operations (read or write), and can be evaluated through:

$$P_{cmt} = \prod_{k=1}^{M} PS_k \tag{9-15}$$

Probability to abort, $P_{abt} = 1 - P_{cmt}$.

Mean Transaction Lifetime

The probability for a transaction to survive exactly l operations ($0 \leq l \leq M$), denoted by a function of l: $\mathrm{f}(l)$, can be derived as:

$$\mathrm{f}(l) = P(lifetime = l) = \begin{cases} (1 - PS_{l+1}) \prod_{k=1}^{l} PS_k \text{, if } 0 \leq \ l < M \\ \prod_{k=1}^{M} PS_k \text{, if } l = M \end{cases} \tag{9-16}$$

Here, the lifetime of a transaction, can be described by the number of completed operations before it aborts (or commits). Mean transaction lifetime, denoted as L, can be derived as:

$$L = \sum_{l=o}^{M} l\mathrm{f}(l) \tag{9-17}$$

Revisiting Lock Holding Time

The original model only considers lock release (either by commit or free operation) as the end of lock holding time. Now we take lock revoke (by an abort) into consideration to compute the mean lock holding time.

$$D_k = \sum_{j=k}^{m} (j-k) R_{op} P_{k,j} + (m-k+1) R_{op} \tilde{P}_{k,m}, \tag{9-18}$$

where $m = \min(k + F, M)$, and $P_{k,j}$ is the probability that the transaction aborts at jth operation, when it requests a lock for its kth operation, and $\tilde{P}_{k,m}$ is the probability that the transaction successfully completes its mth operation, when it requests a lock for its kth operation (in other words, the probability of successful completion between its kth and mth operations).

$$P_{k,j} = (1 - PS_j) \prod_{i=k}^{j-1} PS_i \tag{9-19}$$

$$\tilde{P}_{k,m} = \prod_{i=k}^{m} PS_i \tag{9-20}$$

Numerical Resolution

We can solve the above model using an iterative procedure. Once assigning values to input parameters, the other model parameters can be evaluated via the provided equations, using the results as the input for the next iteration. The desired computational accuracy can be fixed by defining a value ε specifying the maximum difference between values obtained by two consecutive iterations (e.g. if P_{abt_n} is the abort probability at iteration n, then we stop the evaluation when $|P_{abt_n} - P_{abt_{n-1}}| < \varepsilon$ is true). We have empirically observed that it always converges in a few iterations.

Results

We evaluate the numerical results of our analytical model using the configurations summarized in Table 9-1. The listed parameters are assigned with the default valued given in Table 9-1, unless otherwise specified.

We observe decreases in P_{abt} with the introduction of free operations (Figure 9-1). We use the parameter F as a metric for the involvement of free operations. When $F = M - 1$, all locks will be released at the end of the transaction (i.e., no free operations involved), the behavior of our context lock protocol becomes the same as a strict 2-phase locking protocol. As F decreases (i.e., increased involvement of free operations), the mean lock holding time also decreases, resulting in a lowered probability to abort. In the extreme case where $F = 0$, all locks will be released

immediately after the operation completes, resulting in the lowest probability to abort, as indicated by the blue curve in Figure 9-1.

The effect of δ is shown in Figure 9-2. In general, we observe that the probability to abort P_{abt} decreases, as δ increases. However, we can no longer observe this effect when $\delta \leq \sigma$. For example, as shown in Figure 9-2, the blue curve ($\delta = 0$) and the green curve ($\delta = 1$) completely coincide. This can be explained by the relaxation mechanism enabled by δ. Such relaxation reduces the abort rate by exploiting the opportunity to reduce the probability of read-write contentions (i.e., P_{rw}). As Equations 9-12 and 9-14 suggest, P_{rw} is a function of δ and σ. When $\delta \leq \sigma$, $P_{rw} \equiv 1$. Hence, P_{abt} will not be reduced.

The effect of σ is shown in Figure 9-2. We find that the effect of σ is too trivial to be observed when $\sigma > \delta$. This is because of the dominance of read operations in our experiment (write ratio $W = 0.05$). With a low write ratio, the majority of the contentions will be between read and write, whose probability of conflict is always 1 (i.e., $P_{rw} \equiv 1$) when $\sigma > \delta$, as explained previously. As a result, the relaxation effect enabled by σ on write-write contentions (i.e., P_{ww}) is insignificant. That explains why the red curve ($\delta = 2$) and the light green curve ($\delta = 4$) nearly coincide in Figure 9-3. However, with an increased write ratio (as shown in Figure 9-4), we will able to separate these curves as the relaxation effect enabled by σ becomes more significant as the write-write contentions become more frequent. Notice that the high write ratio also results in a more drastic increase in P_{abt} - in Figure 9-4, P_{abt} exceeds 0.9 when λ reaches 0.1 for all four curves.

When $\sigma < \delta$, the probability of read-write conflict P_{rw} decreases as σ decreases. Although a lower σ yields a higher P_{ww}, the overall probability to abort is decreased because of the read dominance in our experiment. That explains why the lowest σ produced the lowest P_{abt} (the blue curve with $\sigma = 0$) in Figure 9-3.

We show the compound effect of σ and δ in Figure 9-5. In this experiment, we use a fixed σ:δ ratio ($\delta = 1.5\sigma$). When $\delta = 0$ and $\sigma = 0$, as indicated by the blue curve in the figure, no relaxation (on δ and σ) is used, resulting in the highest P_{abt}. As σ (and accordingly δ) increases, we observe that P_{abt} decreases. As λ increases, the four curves start to converge. We observe the maximum performance gain when λ is at the interval [0.25, 0.75].

Analysis 2 – τ and θ Analysis

Objective

Based on the model established in the previous analysis, we further analyze the effect of $\{\tau, \theta\}$ in the context lock protocol. Again, we use P_{abt}- the probability to abort as a metric to evaluate the effect of τ and θ.

Probability of a Write Success

For a specific write, its write time can be modeled as: $r_{write} = r_{actuation} + r_{completion} + r_{durability}$, where $r_{actuation}$ represents the actuation time of the device, $r_{completion}$ is a random variable representing the waiting time to attain the desired context after the device actuation, and $r_{durability}$ is the durability window (the time to protect the attained context to be durable). By the definitions of τ and θ, we know that $r_{completion} \leq \tau$, and $r_{durability} = \theta$.

We use another random variable $\tilde{r}_{completion}$ to represent the actual time needed to attain the desired context. Notice that $\tilde{r}_{completion}$ is not bounded by τ - when $\tilde{r}_{completion} > \tau$, it means that τ-window expires before the completion, which leads to an abort of the transaction according to the context lock protocol. We assume the distribution of $\tilde{r}_{completion}$ to be log-normal on a semi-infinite interval $[0, +\infty)$, which has been successfully fitted to empirical reaction times [44]. We denote such distribution by: $\tilde{r}_{completion} \sim \ln \mathcal{N}\ (\mu_\tau, \sigma_\tau^2)$.

For a write access towards i, after its update lock is obtained, the probability of its success is determined by the probabilities of two independent events (assuming the actuator device is failure-free): attaining the desired context within τ-window (*completion*), and persisting the attained context through θ-window (*durability*). We denote the probability of completion as P_τ, and the probability of durability as P_θ. The probability of success for a write is: $P_{w\_suc,i} = P_\tau P_\theta$.

Probability of Completion - $\boldsymbol{P_\tau}$

$$P_\tau = P\left(\tilde{r}_{completion} \leq \tau\right) = CDF_{\ln \mathcal{N}}(\tau; \mu_\tau, \sigma_\tau^2) = \Phi\left(\frac{\ln \tau - \mu_\tau}{\sigma_\tau}\right), \qquad (9\text{-}21)$$

where $\Phi(x)$ is the CDF of the standard normal distribution defined previously.

Mean Time to Completion - $\boldsymbol{R_{completion}}$

$$R_{completion} = \int_0^\tau x PDF_{\ln \mathcal{N}}(x; \mu_\tau, \sigma_\tau^2)\, \mathrm{d}x = \int_0^\tau \frac{1}{\sqrt{2\pi\sigma_\tau^2}} e^{-\frac{(\ln x - \mu_\tau)}{2\sigma_\tau^2}} \mathrm{d}x \qquad (9\text{-}22)$$

After some algebra, we have

$$R_{completion} = \frac{1}{2} e^{\frac{\sigma_\tau^2 + 2\mu_\tau}{2}} \left(\mathrm{erf}\left(\frac{\ln \tau - \sigma_\tau^2 - \mu_\tau}{\sqrt{2\sigma_\tau^2}}\right) + 1\right), \qquad (9\text{-}23)$$

where $\mathrm{erf}(x)$ is the error function of normal distribution defined by

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, \mathrm{d}t. \tag{9-24}$$

Probability of Durability - P_θ

The durability constraint will be violated if a random event takes place that changes the attained context during the θ-window. To model the occurrence of this event, a random variable $\tilde{r}_{durability}$ is defined as the time interval between attaining a desired context (at the end of completion stage) and the occurrence of the event that corrupts the attained context. Similar to the distribution of $\tilde{r}_{completion}$, we assume that $\tilde{r}_{durability}$ also conforms to a log-normal distribution: $\tilde{r}_{durability} \sim \ln \mathcal{N}\,(\mu_\theta, \sigma_\theta^2)$. Hence, the probability that durability is sustained during the θ-window can be evaluated as the probability of $\tilde{r}_{durability} > \theta$. We derive P_θ as following:

$$P_\theta = P\left(\tilde{r}_{durability} > \theta\right) = 1 - CDF_{\ln \mathcal{N}}\left(\theta; \mu_\theta, \sigma_\theta^2\right) = 1 - \Phi\left(\frac{\ln \theta - \mu_\theta}{\sigma_\theta}\right) \tag{9-25}$$

The length of θ-window is a constant, i.e., $r_{durability} = \theta$, according to the lock protocol. Hence its corresponding mean value, $R_{durability}$, is also θ.

Updating Transaction Model with τ and θ

Based on the new variables- P_τ, $R_{completion}$, and P_θ derived above, we update the transaction model established in Analysis 1. We first update R_{write} (mean time to write):

$$R_{write} = R_{actuation} + R_{completion} + R_{durability} \tag{9-26}$$

Then we update PS_k (the probability that the kth operation in a transaction completes successfully). In the case of write operation, the probability of its success is determined by both two independent conditions: (1) free of lock contention, and (2) successful execution (i.e., surviving both τ and θ windows). The probability of condition

(1) is already derived as $1 - PW_{write,i}$. And the probability of condition (2) is defined by $P_{w\_suc,i} = P_\tau P_\theta$. We plug in $P_{w\_suc,i}$ and update the derivation of PS_k.

$$PS_k = \sum_{i=1}^{I} \{A_{i,k} \left[P_{read,i}\left(1 - PW_{read,i}\right) + P_{write,i}\left(1 - PW_{write,i}\right)P_{w\_suc,i}\right]\} \quad (9\text{-}27)$$

Using the updated Equations 9-27 and 9-28, we can analyze the effects of τ and θ by evaluating P_{abt} (the probability to abort) under different values for τ and θ parameters.

Results

Employing the same numerical resolution approach as in Analysis 1, we evaluate the effect of $\{\tau, \theta\}$ in the analytical model.

The effect of τ is shown in Figure 9-6. In this experiment, to focus only on the effect of τ, we assume a write operation always survives its θ-window (i.e., $P_\theta = 1$). The value of τ affects two important variables: (1) the probability of completion for a write operation (i.e., $P_{w\_suc,i}$), and (2) the mean time for a write operation (i.e., R_{write}). As τ increases, its implication is mixed - $P_{w\_suc,i}$ increases with τ, which leads to a lower P_{abt}, and R_{write} also increases with τ, which leads to a higher P_{abt}. When λ is relatively low, the effect of $P_{w\_suc,i}$ is more dominant than the effect of R_{write}. That explains why higher τ yields to lower P_{abt} when λ is close to 0, as indicated by the left part of Figure 9-6. When λ is high, the performance penalty (in terms of abort ratio) caused by large R_{write} overwhelms the performance gains by high $P_{w\_suc,i}$. As a result, curves with higher τ cross over the ones with lower τ, as shown in the right part of Figure 9-6.

The effect of θ is shown in Figure 9-7. In this experiment, to focus only on the effect of θ, we assume a write operation always survives its τ-window (i.e., $P_\tau = 1$), and

we further assume the mean to completion is a constant (i.e., $R_{completion} = 4$). We observe that as θ increases, P_{abt} also increases. This is because the increase in θ would cause both the probability of completion for a write operation (i.e., $P_{w\_suc,i}$), and the mean time for a write operation (i.e., R_{write}) to increase, resulting in a higher overall abort rate P_{abt} observed.

We show the compound effect of τ and θ in Figure 9-8. In this experiment, we use a fixed τ:θ ratio ($\tau = \theta$).

Table 9-1. Input Configurations

Parameter(s)	Meaning	Default value
M	Mean length of pervasive transactions	10
I	Total number of context entities	10
A	Access matrix	A uniform matrix with each element $A_{i,k} = 0.1$
W	Universal write ratio	0.05
R_{read}	Mean time to read	1
R_{write}	Mean time to write (for Analysis 1 only)	10
$R_{actuation}$	Mean time for device actuation (for Analysis 2 only)	2
$\mathcal{N}(\mu_d, \sigma_d^2)$	Distribution of target value d	$\mu_d = 0$, $\sigma_d = 1$
$\mathcal{N}(\mu_c, \sigma_c^2)$	Distribution of current value c	$\mu_c = 0$, $\sigma_c = 1$
$\ln \mathcal{N}\ (\mu_\tau, \sigma_\tau^2)$	Distribution of $\tilde{r}_{completion}$	$\mu_\tau = 2$, $\sigma_\tau = 1$
$\ln \mathcal{N}\ (\mu_\theta, \sigma_\theta^2)$	Distribution of $\tilde{r}_{durability}$	$\mu_\theta = 2$, $\sigma_\theta = 1$

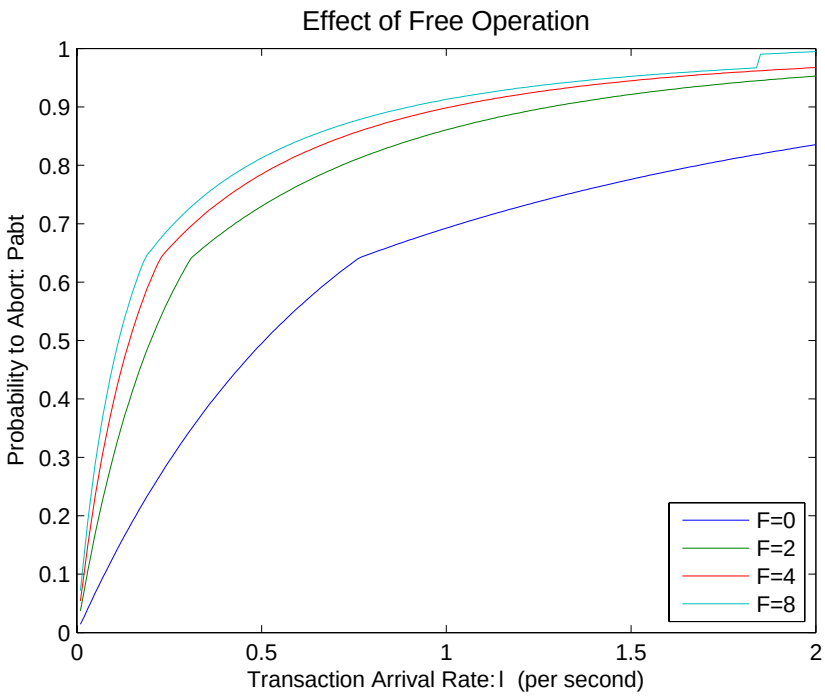


Figure 9-1. Effect of Free Operations on the probability to abort P_{abt}. We show P_{abt} under four different values for F (mean distance to free). In this experiment, $\delta = 0, \sigma = 0$.

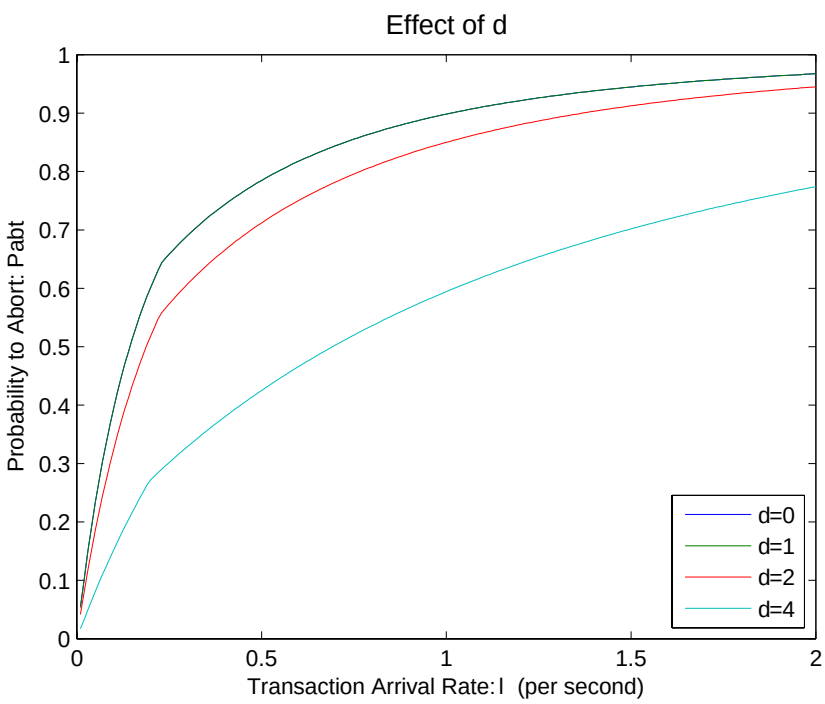


Figure 9-2. Effect of relaxation parameter δ on the probability to abort P_{abt}. We show P_{abt} under four different values for δ. In this experiment, $F = 4, \sigma = 1$.

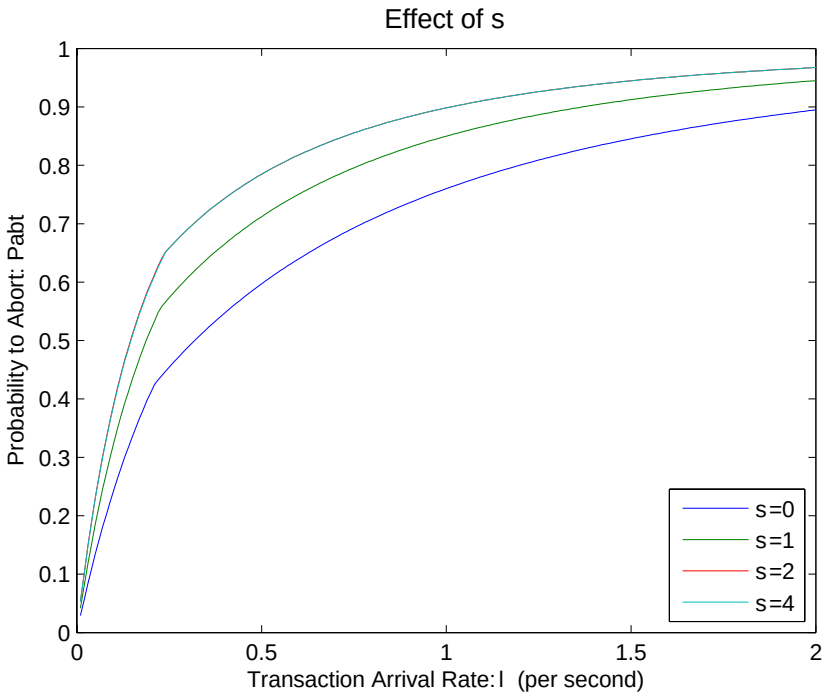


Figure 9-3. Effect of relaxation parameter σ on the probability to abort P_{abt}. We show P_{abt} under four different values for σ. In this experiment, $F = 4, \delta = 1$.

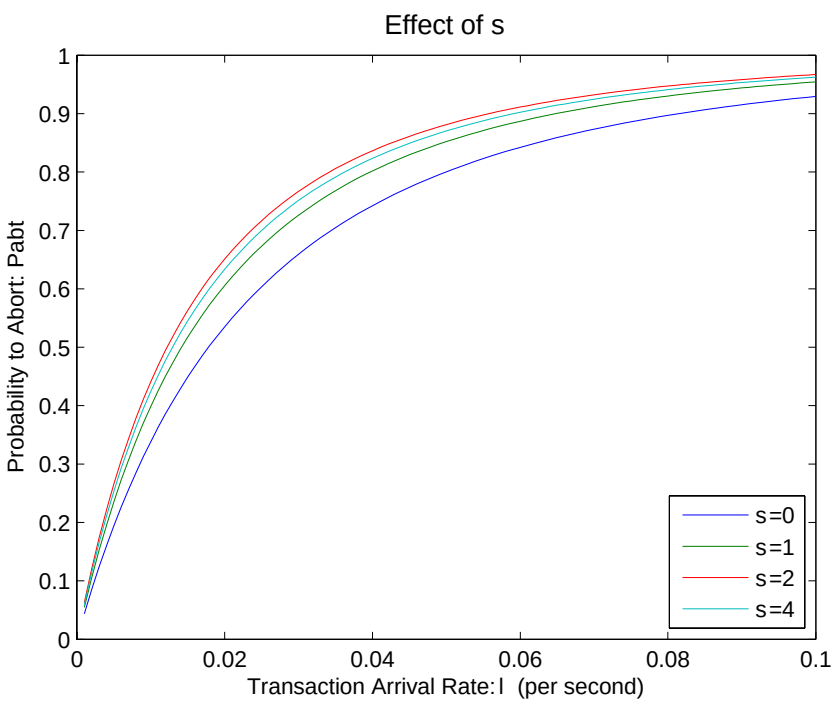


Figure 9-4. Effect of relaxation parameter σ on the probability to abort P_{abt} under a high write ratio. We show P_{abt} under four different values for σ. In this experiment, $F = 4, \delta = 2, W = 0.3$.

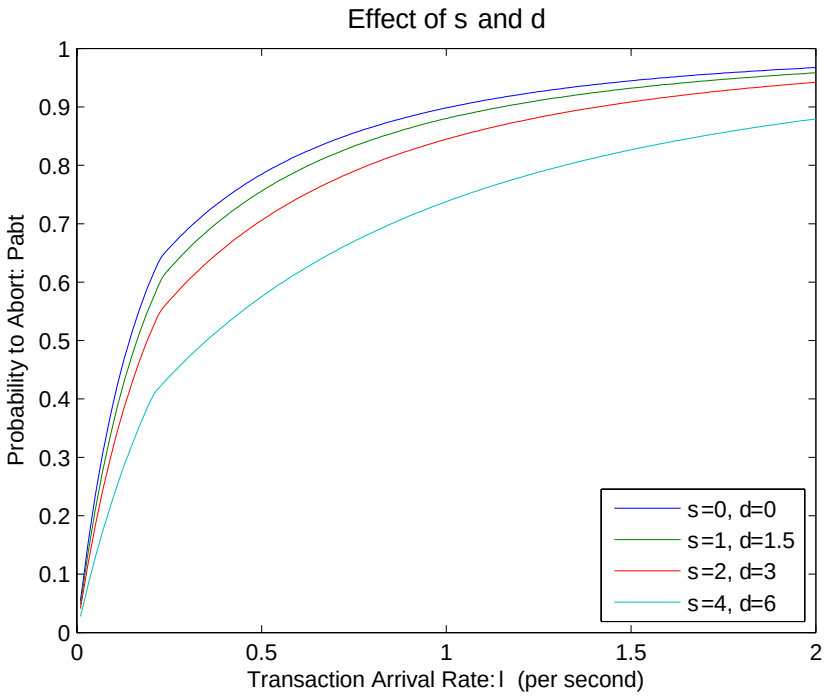


Figure 9-5. Compound effect of σ and δ. We show P_{abt} under four different combinations for (σ, δ) values. In this experiment, $F = 4, \delta = 1.5\sigma$.

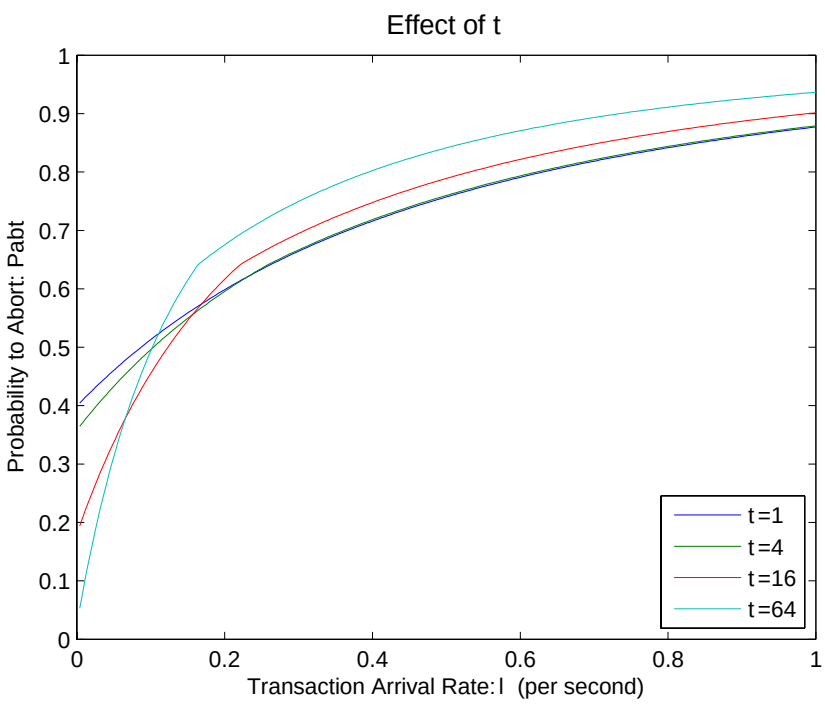


Figure 9-6. Effect of τ on the probability to abort P_{abt}. We show P_{abt} under four different values for τ. In this experiment, $F = 4, \sigma = 0, \delta = 0, R_{durability} = 4, P_{\theta} = 1$.

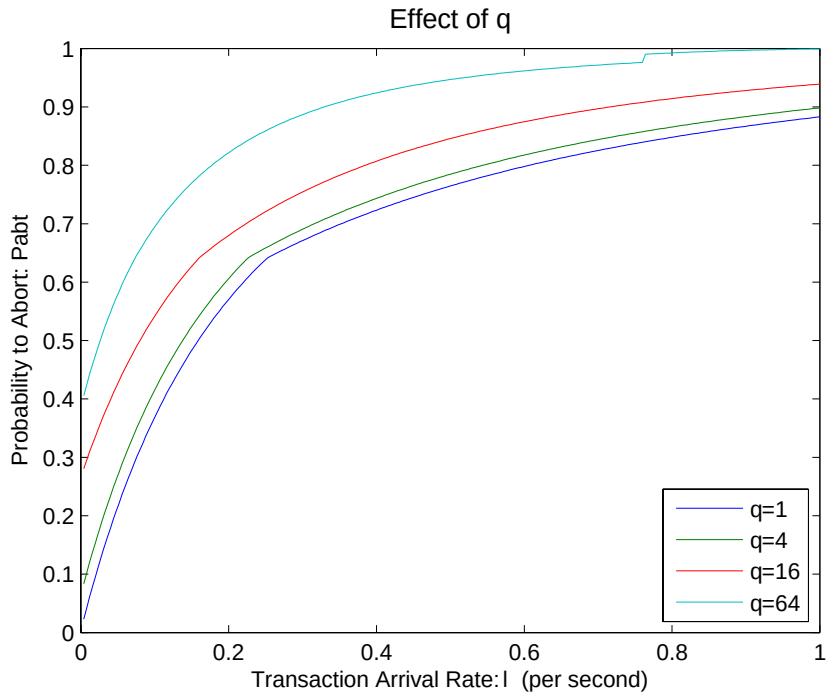


Figure 9-7. Effect of θ on the probability to abort P_{abt}. We show P_{abt} under four different values for θ. In this experiment, $F = 4, \sigma = 0, \delta = 0, R_{completion} = 4, P_{\tau} = 1$.

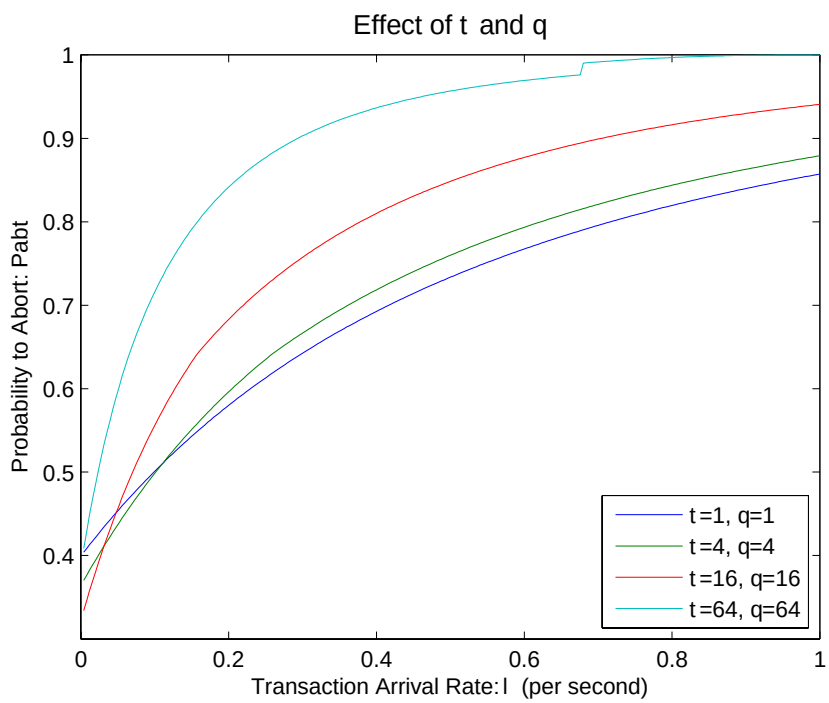


Figure 9-8. Compound effects of τ and θ on the probability to abort P_{abt}. We show P_{abt} under four different combinations for (τ, θ) values. In this experiment, $F = 4, \sigma = 0, \delta = 0$.

CHAPTER 10
CONCLUSION

As computing permeates the physical world through sensors and actuator devices, the consequence of logical errors also extends to the physical realm and creates real safety risks in our everyday life. Traditional safety engineering approach fell short as it lacks the power and flexibility to meet a broad spectrum of safety demand in an open and constantly evolving computing environment. This research presents a safety-oriented programming model, tailored specifically to address the safety challenges in pervasive spaces.

The first challenge is to make pervasive systems understand safety. Hardcoded rules and policies assume fixed risk scenario and hence may become stale as use cases change in pervasive environments. In addition, embedding these rules and polices into various software/hardware artifacts creates a fragmentation of safety knowledge, which may lead to conflicts and uncertainties in the overall system behavior. This research aims to construct a uniform and integrated safety knowledge base, by enabling domain experts, in addition to programmers, to specify safety knowledge and concerns using a domain-independent ontology for safety, we call DiOS. We presented the ontology design of DiOS in Chapter 5, and a case study of DiOS used in Device Description Language (DDL) in Chapter 6, which demonstrated a path to an integrated safety knowledge base by extending domain-specific languages such as DDL with DiOS to facilitate specification of safety constraints for domain experts across application domains.

Once all safety constraints are established in an integrated knowledge base, a challenge follows in that these safety constraints need to be enforced at system

runtime. To formalize the runtime cyber-physical interactions in pervasive systems, we presented pervasive transaction, a novel concept core to our programming model. We demonstrated in our model that the safety criteria can be formalized into a set of runtime properties for transaction processing known as AI^2D (Chapter 7). To deliver assurance to AI^2D-safety, we presented a context lock protocol, which utilizes lock concepts to protect critical context, and to isolate conflicting pervasive transactions. A proof is presented in Chapter 8 that the context lock protocol always produces a safe transaction processing schedule that meets the AI^2D criteria.

Finally, we evaluated the performance of our safety approach in Chapter 9. Safety can only be achieved at a cost. A cost-benefit analysis is important to identify the optimal safety strategy. We used transaction abort probability, $\mathrm{P_{abt}}$, as a key metric in our analysis, and established a probabilistic model that correlates $\mathrm{P_{abt}}$ with other key operation parameters defined in the context lock protocol. We studied and demonstrated the effect of each operation parameters on $\mathrm{P_{abt}}$.

APPENDIX A
DDL DATA MODELS

This appendix presents graph representations for data models of DDL devices. The tree graphs provide a hierarchical view of DDL data models of sensors, actuators and complex devices as shown in Figure A-1. The graph is also a map for all the elements of the DDL language. In the map, each tree node represents an element whose contents are represented by its sub-tree. In addition, we use various symbols to represent the number of occurrences of each element as shown in Table A-1.

Figure A-2 and Figure A-3 in this appendix presents the hierarchical structure of data models for sensors and data models respectively.

Table A-1. Symbols notations for element occurrences.

Notation	Meaning
?	Zero or one
*	Zero or more
+	One or more
{}	Choose one

```
DDL := {Sensor, Actuator, Device}
```

Figure A-1. The data model of the DDL root element.

```
Sensor := (Description, Interface)
Description := (Name, Verbose_Description ?, Physical, Version ?, Device_Type,
Vendor ?, UniqueId, Location)
Physical := (Dimensions ?, Operating_Environment ?)
Dimensions := (Length, Width, Height)
Operating_Environment := (Temperature, Humidity)
Interface := (Signal +, Reading +)
Signal := (Operation, Type, Measurement, Unit, Number, Range)
Range := (Min, Max)
Reading := (Type, Computation, Measurement, Unit)
Computation := (Type, Expression, Range)
```

Figure A-2. DDL Data Model for Sensor.

```
Device := (Description, Interface)
Description := (Name, Verbose_Description ?, Physical, Version ?, Device_Type,
Vendor ?, UniqueId, Location)
Physical := (Dimensions ?, Operating_Environment ?)
Dimensions := (Length, Width, Height)
Operating_Environment := (Temperature, Humidity)
Interface := (Signal +, Reading +)
Signal := (Operation, Type, Method_Name, Parameter +, Frequency,
Permissible_Status)
Parameter := (Parameter_Name, Parameter_Characterization)
Range := (Min, Max)
Reading := (Type, Computation, Measurement, Unit)
Computation := (Type, Expression, Range)
```

Figure A-3. DDL Data Model for Device.

APPENDIX B
XML SCHEMA FOR DDL

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:complexType name="DDL">
      <xsd:sequence>
         <xsd:element name="Sensor" type="Sensor" minOccurs="0" maxOccurs="1" />
         <xsd:element name="Actuator" type="Actuator" minOccurs="0" maxOccurs="1" />
         <xsd:element name="Device" type="Device" minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Sensor">
<xsd:sequence>
  <xsd:element name="Description" type="Description" minOccurs="1" maxOccurs="1" />
  <xsd:element name="Interface" type="Interface" minOccurs="1" maxOccurs="1" />
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Actuator">
   <xsd:sequence>
   <!-- Actuator Type is to be defined   -->
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Device">
   <xsd:sequence>
      <xsd:element name="Description" type="Description" minOccurs="1" maxOccurs="1" />
     <xsd:element name="Interface" type="Interface" minOccurs="1" maxOccurs="1" />
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Description">
<xsd:sequence>
<xsd:element name="Name" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
```

```
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Device_Type" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Physical" />
  <xsd:enumeration value="Virtual" />
    </xsd:restriction>
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Verbose_Description" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Vendor" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Version" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
  <xsd:element name="Physical" type="Physical" minOccurs="1" maxOccurs="1" />
  <xsd:element name="UniqueId" type="UniqueId" minOccurs="0" maxOccurs="1" />
  <xsd:element name="Location" type="Location" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Physical">
<xsd:sequence>
  <xsd:element name="Dimensions" type="Dimensions" minOccurs="0" maxOccurs="1" />
  <xsd:element name="Operating_environment" type="Operating_environment"
     minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="Dimensions">
<xsd:sequence>
<xsd:element name="Length" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:float" />
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Width" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:float" />
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Height" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:float" />
     </xsd:simpleType>
     </xsd:element>
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Operating_environment">
<xsd:sequence>
  <xsd:element name="Temperature" type="Temperature" minOccurs="0" maxOccurs="1" />
  <xsd:element name="Humidity" type="Humidity" minOccurs="0" maxOccurs="1" />
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Temperature">
<xsd:sequence>
  <xsd:element name="Range" type="Range" minOccurs="1" maxOccurs="1" />
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Humidity">
<xsd:sequence>
  <xsd:element name="Range" type="Range" minOccurs="1" maxOccurs="1" />
     </xsd:sequence>
```

```
</xsd:complexType>

<xsd:complexType name="Range">
<xsd:sequence>
<xsd:element name="Max" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:float" />
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Min" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:float" />
     </xsd:simpleType>
     </xsd:element>
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="UniqueId">
   <xsd:sequence>
   <!-- Encoding scheme to be decided   -->
     </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Location">
   <xsd:sequence>
   <!-- Encoding scheme to be decided   -->
   </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Interface">
<xsd:sequence>
  <xsd:element name="Signal" type="Signal" minOccurs="0" maxOccurs="unbounded" />
  <xsd:element name="Reading" type="Reading" minOccurs="0" maxOccurs="1" />
     </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="Signal">
<xsd:sequence>
<xsd:element name="Operation" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Input" />
  <xsd:enumeration value="Output" />
     </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Type" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Analog" />
  <xsd:enumeration value="Digital" />
  <xsd:enumeration value="Protocol" />
  <xsd:enumeration value="Logical" />
     </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Measurement" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Unit" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
     </xsd:simpleType>
     </xsd:element>
<xsd:element name="Number" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Multiple" />
  <xsd:enumeration value="Single" />
     </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
```

```
  <xsd:element name="Range" type="Range" minOccurs="0" maxOccurs="1" />
   <xsd:element name="Method_name" minOccurs="0" maxOccurs="1">
   <xsd:simpleType>
   <xsd:restriction base="xsd:string">
    </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
   <xsd:element name="Parameter" type="Parameter" minOccurs="0" maxOccurs="unbounded"
      />
   <xsd:element name="Frequency" minOccurs="0" maxOccurs="1">
   <xsd:simpleType>
   <xsd:restriction base="xsd:integer">
    </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
   <xsd:element name="Allowed_Status" minOccurs="0" maxOccurs="1">
   <xsd:simpleType>
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="INIT " />
     <xsd:enumeration value="BINDED" />
      <xsd:enumeration value="ALIGNED" />
      <xsd:enumeration value="EXEC" />
      <xsd:enumeration value="TERMINATE" />
      <xsd:enumeration value="EMERGENCY" />
      <xsd:enumeration value="ERROR" />
      <xsd:enumeration value="ALL" />
    </xsd:restriction>
     </xsd:simpleType>
     </xsd:element>
      </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Parameter">
<xsd:sequence>
<xsd:element name="Parameter_name" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
      </xsd:restriction>
```

```
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Parameter_characterization" type="xsd:float"  minOccurs="0"
     maxOccurs="unbounded"/>
    </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required" />
  <xsd:attribute name="enumeration" type="xsd:boolean" use="required" />
  <xsd:attribute name="discrete" type="xsd:boolean" use="required" />
</xsd:complexType>


<xsd:complexType name="Reading">
<xsd:sequence>
<xsd:element name="Type" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Basic" />
  <xsd:enumeration value="Derived" />
  <xsd:enumeration value="Physical" />
    </xsd:restriction>
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Measurement" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Unit" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
  <xsd:element name="Computation" type="Computation" minOccurs="0" maxOccurs="1" />
  <xsd:element name="Range" type="Range" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>
```

```
<xsd:complexType name="Computation">
<xsd:sequence>
<xsd:element name="Type" minOccurs="1" maxOccurs="1">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="Aggregate" />
  <xsd:enumeration value="Formula" />
  <xsd:enumeration value="Map" />
    </xsd:restriction>
    </xsd:simpleType>
    </xsd:element>
<xsd:element name="Expression" minOccurs="0" maxOccurs="1">
<xsd:simpleType>
  <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    </xsd:element>
    </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

APPENDIX C
SAMPLE DDL DOCUMENT FOR TMP36 ANALOG TEMPERATURE SENSOR

```
<?xml version="1.0" encoding="UTF-8" ?>
<DDL version="1.21">
    <Sensor>
        <!-- General information of the device -->
        <Description>
            <!-- Name of the device -->
            <Name>Temperature Sensor </Name>
            <!-- Type of Device (Physical(Singleton) or Virtual) -->
            <Device_Type>Physical</Device_Type>
            <!-- Description of the device -->
            <Verbose_Description>
            TMP36 Analog Temperature Sensor
            </Verbose_Description>
            <!-- Device vendor -->
            <Vendor>University of Florida </Vendor>
            <!-- Device version -->
            <Version>1.0 </Version>
            <Physical>
                <!-- Dimensions of the device -->
                <Dimensions>
                    <!-- Length in mm, left blank if unknown -->
                    <Length>24</Length>
                    <!-- Width in mm, left blank if unknown -->
                    <Width>34</Width>
                    <!-- Height in mm, left blank if unknown -->
                    <Height>24</Height>
                </Dimensions>
                <!-- Permissible environment for operation -->
                <Operating_environment>
                    <!-- Permissible temperature range for operation -->
                    <Temperature>
                        <Range>
                            <!-- left blank if unknown -->
                            <Min></Min>
                            <!-- left blank if unknown -->
                            <Max></Max>
                        </Range>
```

```
            </Temperature>
            <!-- Permissible humidity range for operation -->
            <Humidity>
                <Range>
                    <!-- left blank if unknown -->
                    <Min></Min>
                    <!-- left blank if unknown -->
                    <Max></Max>
                </Range>
            </Humidity>
        </Operating_environment>
    </Physical>
</Description>
<!—Interfaces of a device-->
<Interface>
    <!-- To avoid confusion, ensure Signal id is always alpha-numeric instead of numeric -->
    <Signal id="s1">
        <!-- Value of Operation attribute can be Input or Output -->
        <Operation>Input</Operation>
        <!--  A Signal Type can be: Analog or Digital or Protocol or Logical
              Analog/Digital is a low level collection of pins
              Protocol is a high level interface to a device which has an
              in-built communication protocol (example: AnD Blood Pressure Monitor)
              Logical is high-level device service.
        -->
        <Type>Analog</Type>
        <!-- Value of Measurement attribute can be: ADC, Digital or a string whose value
              is equal to the Reading->Measurement attribute of another physical/virtual sensor
        -->
        <Measurement>ADC</Measurement>
        <Unit>null</Unit>
        <!-- Number can be Single or Multiple (many signal inputs of same type) -->
        <Number>Single</Number>
        <Range>
            <!-- leave blank if you don't know -->
            <Min>0</Min>
            <!-- leave blank if you don't know -->
            <Max>1023</Max>
        </Range>
    </Signal>
```

```
            <Reading>
                <!-- Type can be: Basic(Virtual Sensor), Derived(Virtual Sensor),
                     Physical (Singleton Sensor)
                -->
                <Type>Physical</Type>
                <Measurement>Temperature</Measurement>
                <Unit>Centigrade</Unit>
                <Computation>
                    <!-- Possible value of the Type attribute can be: Aggregate, Formula or Map -->
                    <Type>Formula</Type>
                    <!-- Possible valid Expression attribute values can be:
                       For Type='Aggregate':   Mean, Median, Mode, Max, Min, Sum
                       For Type='Formula': <numerical expression as function of signal ids>
                       For Type='Map': <map of signal ids to range of output values>
                    -->
                    <!-- The formula function conforms to Java syntax  -->
                    <Expression>
                    reading = (((s1/1023)*3.3)-0.5)*(1000/10)
                    </Expression>
                    <!--Range can be calculated automatically -->
                    <Range>
                        <Min></Min>
                        <Max></Max>
                    </Range>
                </Computation>
            </Reading>
        </Interface>
    </Sensor>
</ DDL >
```

APPENDIX D
DATA STRUCTURE DESIGN FOR PERVASIVE TRANSACTION PROCESSOR

```
class TransactionInstance{
        int transactionID;
        Vector<Operation> operations;
        enum submissionStatus;
        enum processingStatus;
        Time arrival;
        Time start;
        Time finish;
}

class TransactionPoolManager{
        int submitTx(); //returns ID of the transaction
        boolean updateTx(int txID);
        boolean closeTx(int txID);
}

class TransactionScheduler{
        enum schedulingStrategy;

        TransactionInstance getNextTx();
        void setStragtegy();
}

class TransactionPool{
        LinkedList<TransactionInstance> transactions;
        TransactionPoolManager pmngr;
        TransactionScheduler ts;
}

class Operation{
        TransactionInstance t;
        Time start;
        Time finish;
        int operationStatus;
        ContextEntity operand;

        boolean isCompatible(Opeartion op);
        boolean isCompatible(ContextLock cl);

}

class Sensing extends Operation{
        double delta;

        boolean onExecute();
}

class Actuation extends Operation{
```

```
        double sigma;

        boolean onExecute();
}

class TransactionProcessor {
        static TransactionPool tPool;
        static ContextLockManager lockManager;
        static ContextMap cMap;
        TransactionInstance currentTransaction;

        void processingCycle();
        void lockSchedule();
        void processOperation();
        void onAbort();
        boolean onCommit();
}

class ContextLockManager {
        static ContextMap cMap;
        static ContextMonitor monitor;

        boolean onReqeust(Operation op);
        boolean onUpgrade(Operation op);
        boolean onDegrade(ContextLock cl)
        boolean onRelease(ContextLock cl);
        boolean onRevoke(ContextLock cl);
        void onViolation(Constraint c, Operation op);
}

class ContextMap{
      Collection<ContextEntity> contextEntities;

}

class ContextEntity{
        HashMap<Reading, Time> cache;
        Collection<ContextLock> lockTable;

        Collection<Transaction> getReaders();
        Collection<Transaction> getWriters();
        Reading updateCurrentVal();
}

abstract class ContextLock{
        ContextEntity protectedEntity;
        int lockStatus;

        int getLockStatus();
        abstract boolean isCompatible(ContextLock lock);
}
```

```
class ReadLock extends ContextLock{
        Sensing sensing;

        boolean isCompatible(ContextLock lock);

}

class WriteLock extends ContextLock{
        Actuation actuation;

        boolean isCompatible(ContextLock lock);

}
```

LIST OF REFERENCES

1. Chen, C., Helal, A. Device Integration in SODA using the Device Integration Language. *In Proc. SAINT 2009*, IEEE, 100-106, 2009. Best Paper Award.

2. Chen, C., Xu, Y., Li, K., Helal, A. Reactive Programming Optimizations in Pervasive Computing, *In Proc. SAINT 2010*, IEEE, 96-104, 2010. Best Paper Award.

3. Yang, H., Helal. A. Safety Enhancing Mechanisms for Pervasive Computing Systems in Intelligent Environment. In: Middleware Support for Pervasive Computing Workshop, held in conjunction with IEEE PerCom 2008, Hong Kong, 2008.

4. Weiser, M, Brown, J.S. The Coming Age of Calm Technology. 1996.

5. Kay, A. Computers, Networks, and Educations. Sci. Am. (Sept. 1991), 138-148.

6. Weiser, M. Some Computer Science Issues in Ubiquitous Computing. *ACM SIGMOBILE Mobile Computing and Communications Review*, Volume 3, Issue 3, July 1999.

7. DSSE Group in University of South Hampton, Safety in Pervasive Computing, Advanced Specialization and Analysis for Pervasive Computing project deliverable report D13. 2003.

8. Occupational Safety and Health Administration, U.S., http://www.osha.gov/

9. The Housing Health and Safety Rating System, U.K., http://www.communities.gov.uk/hhsrs

10. Chen, C. Helal, S. Toward a Programming Model for Safer Pervasive Spaces. *2010 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, 2010. Xi'an, China. pp. 52-57.

11. Chen, C., Helal, S., de Deugd, S. and Smith, A. Roles and System Architecture in the Design, Creation and Management of Smart Spaces. in *proceedings of Workshop on Programming Methods for Mobile and Pervasive Systems*, in conjunction with Pervasive 2010, Helsinki, Finland.

12. Ramamritham, K and Chrysanthis, P. K. Advances in Concurrency Control And Transaction Processing. *IEEE Computer Society*. 1997.

13. Henricksen, K., Indulska, J. and Rakotonirainy, A. Modeling Context Information in Pervasive Computing Systems. *Pervasive 2002*, LNCS 2414, pp. 167-180, 2002.

14. Gray, P. D., Salber, D. Modeling and Using Sensed Context Information in the Design of Interactive Applications. *Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01),* Toronto, Canada, May 2001.

15. Helal, A., Mann, W., Elzabadani, H., King, J., Kaddourah, Y. and Jansen, E. Gator Tech Smart House: A Programmable Pervasive Space. *IEEE Computer magazine,* March 2005, pp 64-74.

16. Analog Devices, Inc. http:// www.analog.com/

17. Life Source, A&D Medical, Inc. http://www.andonline.com/

18. Atlas Middleware, Pervasa, Inc., http://www.pervasa.com/

19. Edge Frontier, Augusta Systems, Inc. http://www.augustasystem.com/

20. Helal, A., Chen, C. The Gator Tech Smart House: Enabling Technologies and Lessons Learned. *Proceedings of the 3rd International Convention on Rehabilitation Engineering & Assistive Technology (i-CREATe '09),* April, 2009. Singapore.

21. Yang, H. Safety-Oriented Programming Models for Pervasive Computing Environments. *Ph.D. dissertation*. 2008.

22. Weiser, M. The Computer for the 21st Century. *Scientific American*, September, 1991.

23. Satyanarayanan, M. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, Vol 8, Issue 4. Aug 2001. pp. 10-17.

24. Chen, C., Bose, R., Helal, A. Atlas: An Open Model for Automatic Integration and Teleprogramming of Smart Objects, *Proceedings of 3rd International Workshop on Design and Integration Principles for Smart Objects (DIPSO'09),* Orlando Oct 2009.

25. Chen, C. Helal. A. Sifting Through the Jungle of Sensor Standards, *IEEE Pervasive Computing*. Volume: 7, Issue: 4. Oct-Dec 2008.

26. Tan, W., et al. Mine Fire Detection System Based on Wireless Sensor Network. *Proc. ICIA 07*, IEEE, 2007, 148-151.

27. Kortuem, G., et al. Sensor Networks or Smart Artifacts? An Exploration of Organizational Issues of an Industrial Health and Safety Monitoring System. *Proc. Int'l Conf. Ubiquitous Computing (Ubicomp 07),* Springer, 2007, 465–482.

28. Zhou, J., et al. Application of Modern Sensor- and Info-Technology in Long-Term Safety Monitoring of Long Span Rigid-Frame Bridge. *Proc. Int'l Conf. ITS Telecommunications*, IEEE, 2006. 37-40.

29. Jiang, C., Peng, J. Research, manufacture and application of GPS-based surveying robot automatic monitoring system for dam safety. *Proc. ICIS 2009*, IEEE, 2009. 151-155.

30. Liu, C., Teng, P. The Safety Design of the Laboratory Monitoring System. *Proc. ICIS 2010*, IEEE, 2010. 373-376.

31. da Silva Sa, J. et al. Monitoring of Temperature Using Smart Sensors Based on CAN Architecture. *Proc. CONIELECOMP 2005*, IEEE, 2005, 218-222.

32. Ramamritham, K and Chrysanthis, P. K. Advances in Concurrency Control And Transaction Processing. *IEEE Computer Society*. 1997.

33. Helal, S. Programming Pervasive Spaces. *IEEE Pervasive Computing*. IEEE 2005. 84-87.

34. Ma, J., et al. Ubisafe Computing: Vision and Challenges (I). *Proc. Autonomic and Trusted Computing*, 2006. Springer 2006. 386-397.

35. Kirillov, I.A., Klimenko, S.V. Plato's Atlantis Revisited: Risk-informed, Multi-Hazard Resilience of Built Environment via Cyber Worlds Sharing. *Proc. Cyberworlds 2010*. IEEE 2010. 445-450.

36. Yao, T. et al. Multi-Feature Fusion Based Outdoor Water Hazards Detection. *Proc. ICMA 2007*. IEEE 2007. 652-656.

37. Yang, H., Chen, C., Bessam, A., Helal, A. A Framework for Evaluating Pervasive Systems, International Journal of Pervasive Computing and Communications (IJPCC), Vol 6 issue 4 (2010), Emerald Press. Nov, 2010.

38. Yang, H. et al. A Context-Driven Programming Model for Pervasive Spaces. In *proceedings of 5th International Conference on Smart Homes and Health Telematics (ICOST'07)*, Springer 2007. 31-43.

39. Gu, T., Pung, H. K. and Yao, J. K. Towards a Flexible Service Discovery. *Elsevier Journal of Network and Computer Applications (JNCA)*. Vol. 28, Issue 3, Elsevier 2005. 233-248.

40. Fitterer, R. and de Witte, B. Enabling Pervasive Healthcare by Means of Event-Driven Service-Oriented Architectures - The Case of Bed Management in Mid-Sized to Large-Sized Hospitals. In *proceedings of 3rd International ICST Conference on Pervasive Computing Technologies for Healthcare*, London.

41. Boukerche, A., Pazzi, R. W. N., Araujo, R. B. A fast and reliable protocol for wireless sensor networks in critical conditions monitoring applications. In *proceedings of 7th ACM international symposium on modeling, analysis and simulation of wireless and mobile systems*, 2004,157-164.

42. Chen, C., Helal, A. A Device-centric Approach to Enhance Safety of the Internet of Things, *International Workshop on Networking and Object Memories for the Internet of Things* (NOMe-IoT 2011). In conjunction with Ubicomp 2011.

43. Cho, E. S., Helal, S. A Situation-based Exception Detection Mechanism for Safety in Pervasive Systems. *Proc. SAINT 2011*. IEEE 2011.

44. Ulrich, R., Miller, J. Information Processing Models Generating Lognormally Distributed Reaction Times, *Journal of Mathematical Psychology*. Vol. 37, Issue 4, December 1993, 513–525

45. Sanzo, P.D., Palmieri, R., Romano, P. Analytical Modeling of Lock-based Concurrency Control with Arbitrary Transaction Data Access Patterns. *Proceedings of WOSP/SIPEW' 10*. Jan, 2010.

46. Leveson, N. *Safeware: System Safety and Computers*. Addison Wesley. 1995.

47. Chen, Z., Motet, G. System Safety Requirements As Control Structures. *Proceedings of COMPSAC'09*. Seattle, WA, USA. July 2009. 324-331.

48. Martins, F., Lopes, L., Barros, J. Towards the Safe Programming of Wireless Sensor Networks. *Journal of EPTCS 17*, 2010. 49-62.

49. Lanzisero, T. Applied Safety Science and Engineering Techniques(ASSET™): Taking HBSE to the next level. In *Proceedings of 2010 Symposium on Product Compliance Engineering (ISPCE 2010)*, Oct 2010. 1-6.

50. Kelly, T., Weaver, R. The Goal Structuring Notation – A Safety Argument Notation. *In Proc. of Dependable Systems and Networks. 2004 Workshop on Assurance Cases*. 2004.

51. Holloway, C. M. Safety Case Notations: Alternatives For The Non-Graphically Inclined*? In Proceedings of 2008 3rd IET International Conference on System Safety.* Oct 2008. 1-6.

52. McDermid, J. A., Pumfrey, D. J. Safety Analysis of Hardware / Software Interactions in Complex Systems. *Proceedings of the 16th International System Safety Conference*, Seattle, WA, USA 1998. 232-241.

53. Railtrack. Engineering Safety Management-The Yellow Book. *Rail Safety and Standards Board*. 2007.

54. Baskar, S. Enforcing safety in pervasive computing environments. *Ph.D. Dissertation*. Perdue University. 2003.

55. de Champs, T. et al. Pervasive safety application with model checking in smart houses: The INOVUS intelligent oven. *In Proceedings of 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. 2011. Seattle, WA. 630-635.

56. The Engineers' Council for Professional Development. *Science*, Volume 94, Issue 2446, 1941. pp. 456.

57. U.S. Bureau of Transportation Statistics (2010) National Transportation Statistics. *Table 2-1: Transportation Fatalities by Mode*. (Report). Retrieved 2010-02-14.

58. Haxthausen, A. E., Peleska, J. Formal development and verification of a distributed railway control system. *IEEE transaction on Software Engineering*, Volume: 26 Issue:8, 2000. 687 – 701.

59. Kletz, T. Human problems with computer control. *Plant/Operations Progress*, vol. 1, no. 4, 1982.

60. Chetan, S., Ranganathan, A., Campbell, R. Towards fault tolerance pervasive computing. *Technology and Society Magazine*, IEEE, 24(1):38--44, Spring 2005.

61. Xia, J., Chang, C., Kim, T., Yang, H., Bose, R., Helal, A. Fault-resilient Ubiquitous Service Composition. *Proceedings of 3rd IET International Conference on Intelligent Environments (IE'07),* Ulm University, Germany, September 24-25, 2007.

62. Khalid, A., et al. Survey of Frameworks, Architectures and Techniques in Autonomic Computing, *Fifth International Conference on Autonomic and Autonomous Systems*, 2009.

63. P. Hu, J. Indulska, R. Robinson, An Autonomic Context Management System for Pervasive Computing, *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*. 2008. 213-223.

64. Kulkarni, D., Tripathi, A.: A Framework for Programming Robust Context-Aware Applications. *IEEE Transactions on Software Engineering*, vol. 36, no.2, IEEE Computer Society Press, 2010.

65. Damasceno, K., Cacho, N., Garcia, A., Romanovsky, A., Lucena, C. Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case. *Software engineering for large-scale multi-agent systems*, ACM New York, 37 – 44, 2006.

66. Augusto, J. C., Mccullagh, P. J., Carlos, J., & Paul, A. Safety Considerations in the Development of Intelligent Environments. *Ambient Intelligence - Software and Applications*, 92, 197-204. 2001.

67. Gabriel, P., Bovenschulte, M., Hartmann, E., Groß, W., Strese, H., Bayarou, K. M., Haisch, M., et al. *Pervasive Computing: Trends and Impacts*. SecuMedia. Retrieved from http://www.bsi.de/literat/studien/percenta/Percenta_elay.pdf. 2006.

68. Ma, J., et al. Ubisafe Computing: Visiosn and Challenges (I). In *Proceedings of Autonomic and Trusted Computing* (ATC). 386-397. 2006.

69. Gupta, S., Chen, Ke-Yu., Reynolds, M., Patel, S.N. LightWave: Using Compact Fluorescent Lights as Sensors. *UbiComp 2011*, Beijing, China, September 17-21, 2011.

70. Cohn, G., Gupta, S., Froehlich, J., Larson, E., and Patel, S. GasSense: Appliance-Level, Single-Point Sensing of Gas Activity in the Home. *Proceedings of Pervasive 2010*, Helsinki, Finland, May 17-20, 2010.

71. Patel, S.N., Reynolds, M.S., Abowd, G.D. Detecting Human Movement by Differential Air Pressure Sensing in HVAC System Ductwork: An Exploration in Infrastructure Mediated Sensing. *In the Proceedings of Pervasive 2008*. Sydney, Australia. 2008. pp 1-18.

72. Larson, E.C., Goel, M., Boriello, G., Heltshe, S., Rosenfeld, M., Patel, S.N. SpiroSmart: Using a Microphone to Measure Lung Function on a Mobile Phone. *Proceedings of the 14th International Conference on Ubiquitous Computing (UbiComp 2012)*, Pittsburgh, USA, Sep 5-8, 2012.

BIOGRAPHICAL SKETCH

Chao Chen received a Bachelor of Science degree in Computer Science from Nanjing University (Nanjing, China) in 2006. In the same year, he was admitted to the Ph.D. program at the University of Florida. His research focuses on programming models to support automatic integration and safety in pervasive systems. He received his Ph.D. from the University of Florida in the fall of 2012.