Copyright 2005

by

Erwin Jansen
I dedicate this work to Miss Lee, the lady who dances in my dreams.
ACKNOWLEDGMENTS

I thank Dr. Sumi Helal for his guidance and support throughout the years that I have been a part of his lab. Without his encouragement, guidance, and belief in success I would not have come this far. I would also like to thank Professors P. Fishwick, M. Bermudez, G. Ritter, and B. Mann for their valuable suggestions. I thank Mr. Bowers for making all these many last-minute arrangements. I thank Kelly Zoboroski for her love, support, and the joy in her heart that she shares with me on a daily base. Without her I doubt if I ever would have started on this endeavor. Finally, I thank my family for letting me go to this far-away country to pursue my dreams.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
</tbody>
</table>

## CHAPTER

1 INTRODUCTION AND MOTIVATION .................................................. 1
   1.1 Limitations of First-Generation Spaces ............................... 2
   1.2 Requirements for Hardware Integration ............................... 3
   1.3 Programmability of a Space ........................................... 4
   1.4 Safety of Spaces ..................................................... 4
   1.5 Behavior versus Information ......................................... 7
   1.6 Goals ................................................................. 7

2 RELATED WORK ........................................................................ 8
   2.1 Smart Spaces ............................................................. 8
      2.1.1 Context-Aware Computing ....................................... 9
      2.1.2 Middleware Architectures ....................................... 11
   2.2 Ontologies and Semantic Web ......................................... 12
      2.2.1 Resource Description Framework ............................... 13
      2.2.2 Describing Domain Knowledge ................................... 14
      2.2.3 Description Logics and the World-Wide Web .................. 16
   2.3 Temporal Logic ............................................................ 16
      2.3.1 Interval Temporal Logic ......................................... 17
      2.3.2 Linear Temporal Logic ........................................... 17
      2.3.3 Computational Tree Logic ....................................... 18

3 PROBLEM DESCRIPTION .......................................................... 19
   3.1 Non Scalable Integration ................................................ 19
   3.2 Closed World Assumption ............................................... 20
   3.3 Fixed-Point Concepts ..................................................... 20
   3.4 Objectives ..................................................................... 21
      3.4.1 Enhanced Programmability ....................................... 21
      3.4.2 Interoperability and Extensibility ......................... 21
      3.4.3 Capability of Exception Capturing ........................... 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Approach</td>
<td>22</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Knowledge Engineering</td>
<td>24</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Software Engineering</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Advantages</td>
<td>26</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Explicitness</td>
<td>26</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Interoperability and Extensibility</td>
<td>27</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Conflict Detection</td>
<td>27</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Capture Environmental Effect</td>
<td>27</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Automatic Tuning and Machine Learning</td>
<td>28</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Scalability</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Obtaining Information from Sensors</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Deriving Context from Sensors Readings</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Using Ontology</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Temporal Context</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Syntax of Temporal Context</td>
<td>35</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Semantics of Temporal Context</td>
<td>37</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Describing Events using Temporal Context</td>
<td>37</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Computational Complexity of Temporal Context</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>Describing Actuators and Services</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Programming the Space</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Beliefs-Desires-Intentions of Users</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>Inheritance and Override</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Scenario and Programming Procedures</td>
<td>44</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Lightening Control Application</td>
<td>45</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Describing Context</td>
<td>45</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Associating Behavior</td>
<td>46</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Connecting Sensors</td>
<td>47</td>
</tr>
<tr>
<td>7</td>
<td>Integrated Development Environment</td>
<td>50</td>
</tr>
<tr>
<td>7.1</td>
<td>Design of the Context Graph</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>Interpretation of Sensor Readings</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Desired Behaviors under Each Context</td>
<td>51</td>
</tr>
<tr>
<td>7.4</td>
<td>Mapping Physical Sensors and Actuators to the Context Model</td>
<td>52</td>
</tr>
<tr>
<td>7.5</td>
<td>Implementation of the IDE</td>
<td>53</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Premise and Platform Selection</td>
<td>53</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Requirement Analysis</td>
<td>54</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Enabling Middleware Runtime</td>
<td>54</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>Safety versus programmability</td>
<td>6</td>
</tr>
<tr>
<td>2–1</td>
<td>Knowledge network</td>
<td>15</td>
</tr>
<tr>
<td>3–1</td>
<td>Negative interaction between entities</td>
<td>25</td>
</tr>
<tr>
<td>4–1</td>
<td>Indoor and outdoor sensors</td>
<td>34</td>
</tr>
<tr>
<td>4–2</td>
<td>A sequence denoting someone entered the house</td>
<td>35</td>
</tr>
<tr>
<td>4–3</td>
<td>Actual sequence of events</td>
<td>38</td>
</tr>
<tr>
<td>6–1</td>
<td>A sample context graph</td>
<td>49</td>
</tr>
<tr>
<td>7–1</td>
<td>Interaction between components</td>
<td>55</td>
</tr>
<tr>
<td>7–2</td>
<td>Entity definition view</td>
<td>57</td>
</tr>
<tr>
<td>7–3</td>
<td>Context browser</td>
<td>58</td>
</tr>
<tr>
<td>7–4</td>
<td>Impermissible context verification</td>
<td>60</td>
</tr>
<tr>
<td>A–1</td>
<td>Open world versus closed world</td>
<td>69</td>
</tr>
</tbody>
</table>
Our aim was to establish a programming model that can be used by developers who build pervasive applications. Currently there is little to no support for programming pervasive spaces. Every solution is still an ad-hoc solution.

Our programming model acknowledges a relationship among sensors, actuators, and software. By making this relationship explicit, we derived a programming model that is geared toward describing knowledge and defining goals. A goal can be accomplished by invoking a series of actuators.

We divide the software-engineering process into two distinct phases. In one phase we are concerned with describing behavior. An engineer specifies how a service operates to achieve a particular goal. The other phase is the description of what the service tries to accomplish.

It also allows us to reason beforehand about the effect of the invocation of a set of services. This in turn helps a developer determine whether two services intend to accomplish similar goals. This is similar to type-checking, which greatly reduces programming mistakes.
The vision of this technology is that, in the future, you would buy an actuator, or software service, that contains a standardized description of its intentional effect on the world. With this description, the house can use the service to avoid undesirable contexts. This directly ties in with our Self-Sensing Spaces and Smart Plug work.
CHAPTER 1
INTRODUCTION AND MOTIVATION

Interest in the area of ubiquitous computing has grown in the last few years. The idea behind ubiquitous computing is that the technology should be calm [1], meaning that many computers are available throughout the physical environment, making them effectively invisible to the user [2]. Hence ubiquitous computing is sometimes called pervasive computing. It is considered the third wave of computing.

Ubiquitous computing is characterized by the "anywhere, any time, any means" paradigm. Anywhere, at any time, and by any means, a user should be able to perform computations that help the user to accomplish his or her goals. For this to happen many different technologies come together. Hardware must provide interaction with the environment (bluetooth, GPS, etc.), and software is needed that interacts and coordinates with the hardware (OSGi, Jini, CORBA, SOAP, WSDL, etc.) [3, 4, 5].

We consider a smart space to be an ambient [6], in the sense that there is a boundary that determines what is part of the smart space, and what is not. Inside the smart space, computations take place, to assist the users of the space. A pervasive space consists of a collection of devices and software that controls these devices. Devices that sense the environment are called sensors. Devices that can change the environment are called actuators.

Sensors sense a particular value in a particular domain, and provide information on a very detailed level. A temperature sensor might tell us that it is 95°F, or a light sensor might tell us that there is 10,000 lux of light coming through the window. However, we are generally not interested in the direct sensor
values, but rather in a high-level description of the state. We want to know that the weather is "hot" and "sunny" outside. The states "hot" and "sunny" encompass a range of temperature and luminescence values. Hard-coding behavior for each possible combination of relevant sensor values is difficult to implement, difficult to debug, and difficult to extend. Working with generalizations is much easier.

Actuators are physical devices with which we can interact. An actuator is capable of changing the state of the world. Sensors, in turn, may observe the effect of an actuator. A light sensor may observe that the smart house (or the resident) turned on a lamp. Based on the observed state of the world, the house or the resident might decide to activate an actuator.

Most research projects that involved ubiquitous computing have been pilot projects that show that pervasive computing is usable. The Active Badge [7] system used badges that transmitted IR-signals to track the location of various persons. This location information was then used to route calls for persons to the nearest phone.

1.1 Limitations of First-Generation Spaces

The majority of pervasive computing research so far has focused on system integration. Most of these are pilot projects demonstrating that pervasive computing is useable (overview [8]). In general, these pilot projects represent ad-hoc, specialized solutions that are not easy to replicate.

Unfortunately, many of these first-generation pervasive computing systems lack the ability to evolve as new technologies emerge or as an application domain matures. Integrating numerous heterogeneous elements is mostly a manual, ad hoc process. Inserting a new element requires researching its characteristics and operation, determining how to configure and integrate it, and tediously and
repeatedly testing to avoid causing conflicts or indeterminate behavior in the overall system.

Integration of new devices does not scale well with this approach. In the worst-case scenario the adding of a new device means reconfiguring \( n \) existing devices. The environments are closed, limiting development or extension to the original available devices or software entities.

1.2 Requirements for Hardware Integration

The devices themselves must be able to integrate and adapt to the relevant circumstances in the space. For devices to be able to integrate there must be a uniform way to interact with the devices. This uniform way of interaction is accomplished by a middleware layer that creates abstractions for the various available devices. Heterogeneous devices can now be accessed in a uniform way. The requirements for the middleware layer are as follows,

- **Open**: A wide range of diversity inherent in the future pervasive computing environments, the openness is one of key requirements. The environment is made up of tiny sensors, mobile devices, and full-fledged computers from diverse sources. Therefore, the middleware architecture must be open and flexible to embrace a variety of entities without any special favor towards particular participants or target domains.

- **Extensible**: The middleware should be able to support a dynamic build-up of middleware itself, new middleware components can be incorporated, components can be upgraded or replaced and unnecessary components can be removed from the middleware. All these changes must be possible without requiring down-time for the whole middleware. The middleware itself evolves over time, and it is can cope with new technologies and requirements.

- **Adaptive**: The middleware should be able to adapt to the changing environments. In other words, the middleware should reconfigure itself
in accordance with changes in the space (through the above component addition, upgrade, and replacement); and should provide a mechanism by which applications can adapt themselves to these changes.

- **Self-integrative:** The middleware should enable a joining entity to explore the middleware environment and support a mechanism by which it can participate in the middleware. In other words, the zero-configuration style support is necessary to automate the joining process.

### 1.3 Programmability of a Space

The middleware layer alone is not sufficient. The whole aim of this exercise is to provide a programming model for software engineers. The programming model determines how engineers will interact and develop software for the middleware layer. A programming model allows complex systems to be understood and their behavior predicted within the scope of the model.

On top of the programming model we aim to build an interactive (or integrated) development environment (IDE). The IDE needed to be a system for supporting the process of writing software for pervasive space. Such a system may include a syntax-directed editor, graphical tools for program entry, and integrated support for compiling and running the program and relating compilation errors back to the source. The goal was to give programmers a clear model and visual tools for programming the actual pervasive space.

### 1.4 Safety of Spaces

In traditional computing systems, such as the desktop pc, there is hardly any interaction with physical devices. In general there are few input devices, and few output devices (I/O devices). A monitor, speakers, keyboard, and mouse sums up a traditional desktop system.

The problem of modeling I/O devices is that they have side effects: specifically they change the state of the system, unlike referentially transparent expressions.
Furthermore, I/O operations are not guaranteed to succeed. Failures can occur, and dealing with these failures is awkward. Modern programming languages provide for exceptions to deal with these exceptional cases. Since few I/O devices affect the actual world, there is no serious concern about conflicting effects on the physical world.

However, as soon as we start to introduce more devices in a space controlled by computers, this does become an issue. A disastrous example would be a computer that decides to turn off the sprinkler system in case of a fire.

Software components are regularly in conflict with one another. A very well known recent example is that the installation of Netscape 8 will break Internet Explorer. Users accept these software flaws with little complaints. Users are unlikely to accept the sprinkler error. Unexpected behavior, or software errors, are far less acceptable in pervasive spaces, especially when they negatively affects the quality of life.

A pervasive space is safe if we are certain that the space will never move to an undesirable state because of an automated invocation of an actuator. The computer should never perform an action that leads to an undesirable state. Since a computer can only change the state of the space using actuators we can make the following divisions in safety:

- **Non-pervasive:** A space that does not contain any computational devices controlling the environment. Every change in the state of the environment is because of to the activity of a living entity.

- **Observational Pervasiveness:** A space that only contains sensors. The sensors provide information about what is taking place inside the space, but nothing will be done to alter the state of the space.

- **Guarded Pervasiveness:** Apart from sensors the space contains actuators that can change the state of the space. The actuators are limited as follows:
If the actuator fails to operate, the space does not change. An example is a lamp. If the light-bulb is broken, turning on the lamp will not change the state of the space.

The space can accurately predict the next state before this state arises. For example if we turn on the lamp, we can predict that there will be light.

The space can derive if a service will activate an actuator. Every software component is capable of reporting which actuator it will activate and under which circumstances.

- **Unguarded Pervasiveness**: The space contains sensors as well as actuators. However, the space is not able to determine whether or not a service will invoke an actuator. The space is not always capable of predicting the next state of the space.

This division has consequence of the power of expression that the language we are using has. The requirement to predict the possible outcome of an action within a limited amount of time has implications for the expressiveness of the programming language. If it takes $O(2^n)$ to determine the next state of the space, it is unlikely the space will be capable to predict the next state of the space in a timely fashion.

![Figure 1–1: Safety versus programmability](image-url)
To guarantee safety of the pervasive space we consider a programming model that is expressive enough to write programs, yet restricted so we can guarantee that state of the space is not undesirable.

1.5 Behavior versus Information

Our approach to guarantee safety is by making use of descriptions of the various available software and hardware components. We describe components using a temporal ontology. This temporal ontology is used to interpret the state of the space and to describe how service change the state of the space.

We separate the development process of software into two phases. The first phase deals with traditional software engineering. The other phase deals with knowledge engineering. In the software engineering phase we describe behavior whereas in the knowledge engineering phase we describe information of relevance.

1.6 Goals

The aim of our research was to establish a programming model that can be used by developers that build pervasive applications. Currently there is little to no support for programming pervasive spaces. The area is still in the stage where every solution is ad-hoc.

The programming model we introduce is based on the relationship between sensors, actuators and software. By making this relationship explicit we can derive a programming model geared towards describing knowledge and defining goals. A goal can be accomplished by invoking a series of actuators.

It also allows us to reason beforehand about the effect of the invocation of a set of services. This in turn helps a developer determine whether two services intend to accomplish similar goals. This is similar to type-checking, which greatly reduces programming mistakes.
CHAPTER 2
RELATED WORK

In recent years ubiquitous computing has received an increasing amount of
attention in the academic community. Here we will review some of the work that
has received a lot of attention over the last decade.

2.1 Smart Spaces

The PARCTAB [9] system consists of palm-sized mobile computers that
can communicate wirelessly through infrared transceivers to workstation-based
applications. The system provided location information to the user, helped to find
resources and acted as a remote control for various devices.

The Active Badge [7] is a system that keeps track of the location of a person
using badges that transmitted an IR-signal. The telephone receptionist could find
out where a person was and direct the call to an appropriate phone. Many consider
the PARCTAB and Active Badge projects to be the first examples of pervasive
computing. These projects showed how computation can be “calmly” integrated in
the environment.

Microsoft’s Easy living project [10] focuses on home and work environment.
The easy living room is aware of the location of people in the room and uses this
information for access to computational devices. The space recognizes users and
gestures and uses this information to adjust the state of the room accordingly.

Pearl [11] is a nursing robot that assists elderly with mild cognitive
impairments. It has an automatic tracking and detection system for persons. The
system reminds the elder of events and can guide them through the environment.

The Roomware and i-Land [12, 13] projects aim to integrate information
technology into room elements as, walls, doors, and furniture. Roomware
components are interactive and networked; some of them are mobile due to independent power supply and wireless networks, and are provided with sensing technology. The focus on these projects seem to lean more towards the hardware aspect of pervasive computing.

Context aware computing is a computing paradigm in which applications can discover and take advantage of contextual information. This could be temperature, location of the user, activity of the user, etc. The definition of context is rather vague and various researchers give different definitions of context. In this chapter we will examine the various definitions and define the key ingredients of context awareness.

2.1.1 Context-Aware Computing

Context is a concept that has been hard to grasp. In general we can consider that context uses external information to identify the current meaning of an object. In language for example context is used to disambiguate certain words [14], whereas in vision recognition it is used to fill in blanks in interpretation [15]. In the area of ubiquitous computing there are many different definitions in use.

Early researchers in the field define context by the following characteristics: a user’s location, environment, identity and time [16, 17]. This view on context is rather application specific. Abowd et al, [18] defined context as follows:

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.

Similarly Chen and Kotz [8] defined context as:

The set of environmental states and settings that either determines an application’s behavior or in which an application event occurs that is interesting to the user.
They also make a distinction between active and passive contexts, where the first influences the behavior and the second is relevant, but not critical.

In the last decade we have seen various projects that take the first steps at context aware computing in the field of pervasive computing [19, 20]. Many of these applications were pilot projects that demonstrated the use of context and used to investigate the usage, limits and benefits of context aware computing. Due to the success of these applications a need arose for a framework for context aware computing.

The context toolkit [21, 22] is a java toolkit that addresses the distinction between context and user input. The context consists of three abstractions: widgets, aggregators and interpreters. Context widgets encapsulate information about a single piece of context, aggregators combine a set of widgets together to provide higher level "widgets" and interpreters interpret both of these. An interpreter for example can use the identity and location widget to derive that "Dr. Helal is in his office". The aim of the toolkit is to hide the complexity of obtaining sensor information. The context toolkit has proven its value in various applications that have been built with it [23, 24, 25].

iQueue [26] and iQL [27] are a programming model and language for composing higher level data from low-level raw data. The difference between iQueue and iQL is that the first one is an implementation of the model in java, whereas the latter is a specific programming language resembling a functional language.

The programming model is based upon so called composers that contain a current value computed from input values. These values in term stem from various data sources (i.e., sensors), or other composers. The runtime system then advertises these composers to external sources.
Every source can either act passively, producing data only on request, or actively producing data whenever new data is available. Using these composers we can construct an acyclic di-graph, often referred to as context graph, where the nodes contain functions that produces values for the nodes higher up in the graph.

Similar to the approach described above is the Solar framework [28, 29, 30]. Solar allows resource to advertise context sensitive names and for applications to make context-sensitive name queries. The main aim of the frameworks is that resources can advertise their existence, that others can find the resources and that context-aware applications may indicate how to derive their desired context from these sources.

Solar provides a similar mechanism as iQL for the derivation of high-level data from low level sources. On top of this they provide a querying mechanism for retrieving context information.

DFuse [31] is another example of a technology that allows the construction and evaluation of a context graph. The DFuse architecture is geared towards mobile devices optimizing it for power consumption.

2.1.2 Middleware Architectures

GaiaOS [32, 33] is a component based middleware operating system running on top of an existing operating system. The GaiaOS provides a set of basic services, such as discovery, security, as well as an application model. The application model provides a mechanisms to build applications for ubiquitous computing scenarios. GaiaOS is based upon the Model-View-Controller [34] paradigm. However, instead of the traditional notion of a controller gaia makes use of a component adapter that is responsible for adapting the format of the model data to an arbitrary output device. Gaia makes use of a scripting language, called lua [35] to do service composition and event handling. Gaia is intended to be a general computational environment.
The SOCAM architecture [36] is a middleware layer that makes use of ontologies and predicates. There is an ontology describing the domain of interest. By making use of rule based reasoning we can then create a set of rules to infer the status of a an entity of interest. Their research shows that ontologies are usable but require some computation power and hence are not suitable for smaller devices.

The CoBra architecture [37] is a broker center agent architecture. At the core is a context broker that builds and updates a shared context model that is made available to appropriate agents and services.

The MavHome project [38, 39] views the smart home as an intelligent agent that perceives its environment through the use of sensors, and can act upon the environment through the use of actuators. The home has certain overall goals, such as minimizing the cost of maintaining the home and maximizing the comfort of its inhabitants. In order to meet these goals, the house must be able to reason about and adapt to provided information. Their approach is based on machine learning and optimization techniques.

2.2 Ontologies and Semantic Web

Most of the information that is accessible through the world wide web is in the html format. This format is geared towards the layout of information on a screen. It does not contain information about the meaning of the document that is being displayed. This is currently a big obstacle in retrieving information on the web. Much effort is being put in developing the so called “semantic web”. According to [40]:

“The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.”

The vision is that we associate meaning with documents in such a way that it can be processed by machines. In order to accomplish this we need a mechanism
to describe information. This can be done by associating meta information with the documents that we store, or by providing an ontology that gives meaning of the concepts used in a particular domain.

We are trying to create a model of the world in symbolic structures that can be processed by a computer. We call this a conceptual model of the world. Conceptual models are well know in the field of computer science, ranging from the entity relationship model [41] to UML [42]. These conceptual models provide abstraction mechanisms that result in a structured information model [43].

The semantic web builds extensively on XML (a simple extensible mark up language), RDF (an XML language to describe resources) and OWL (a RDF language to describe ontologies). In the following subsections we will investigate these various technologies. We assume the reader is familiar with XML [44].

2.2.1 Resource Description Framework

The Resource Description Framework [45] (RDF) is a language for representing information about resources in the World Wide Web. By generalizing the concept of a "Web resource", RDF can also be used to represent information about things that can be identified on the Web, even when they cannot be directly retrieved on the Web.

The idea behind RDF is that we can describe resources by making statements about these resources. The statements that we make consists of properties with values that describe a particular resource. This is done by associating two objects together with a predicate. The part that we are describing is called the subject. It is associated with another object through a predicate, that describes the subject. Take for example the following English sentence:

http://www.cise.ufl.edu/people/ejansen has a creator Erwin Jansen

We can see the following relationships:

- the subject is the URL http://www.cise.ufl.edu/people/ejansen
- the predicate is the word "creator"
- the object is the phrase "Erwin Jansen"

RDF can be used to describe any relationship we want using the appropriate predicate. These descriptions are given in an XML language, and hence are easily processed by computers.

However the freedom RDF gives introduces a new problem. How are we going to pick appropriate predicates and how do we interpret these predicates. Anyone can make arbitrary predicates, but we do not have any computational guarantees whatsoever that we are able to deduce any valuable information from our descriptions.

Drawing conclusions from a set of rdf statements might not be feasible in a limited amount of time. Although RDF is a good start for describing resources, it leaves us with too much freedom. We need a way of describing knowledge with better computational guarantees, such as a description logic.

2.2.2 Describing Domain Knowledge

A mechanism for describing knowledge are so called description logics \[46\]. Which we shall briefly discuss here.

Description logics are knowledge representation languages tailored for expressing knowledge about concepts and concept hierarchies. They are usually given a Tarski style declarative semantics, which allows them to be seen as sub-languages of predicate logic. They are considered an important formalism unifying and giving a logical basis to the well known traditions of semantic networks, object-oriented representations, semantic data models, and type systems.

The basic building blocks are concepts, roles and individuals. Concepts describe the common properties of a collection of individuals and can be considered as unary predicates which are interpreted as sets of objects. Roles are interpreted
Figure 2–1: Knowledge network

as binary relations between objects. For example in figure 2–1 there is the role of a parent, expressed through the hasChild relationship.

Each description logic defines also a number of language constructs (such as intersection, union, role quantification, etc.) that can be used to define new concepts and roles. We could express the concept of a father as being a parent that is not a woman.

Description logics can be translated into a subset first order predicate logics. For example the expression Person and Woman can be translated into

\[ Person(x) \land Woman(x) \]

where \( x \) is a variable that ranges over all the individuals in the interpretation domain and \( C(x) \) is true for all \( x \) that belong to context \( C \).

Note that first order logic (FOL) is not decidable, but that description logics are a subset of fol that is decidable.

The main reasoning tasks are classification and satisfiability, subsumption and instance checking. Subsumption represents the is-a relation. Classification is the computation of a concept hierarchy based on subsumption.

There is a tradeoff we must make between the expressiveness of our description logic and the time complexity of deriving knowledge. The more expressive our
language is, the more time it will take to compute whether or not a sentence is valid [47].

2.2.3 Description Logics and the World-Wide Web

OWL [48] is a description logic that based on RDF. Using OWL we can describe an ontology. There are three versions of owl, each with different computational guarantees.

- **OWL Lite**: Owl lite is primarily used for defining a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. Owl Lite has a lower formal complexity than OWL DL.

- **OWL DL**: Has maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions. OWL DL is a description logic.

- **OWL Full**: OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

2.3 Temporal Logic

In logic, the term temporal logic is used to describe any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. It is sometimes also used to refer to tense logic, a particular modal logic-based system of temporal logic introduced by Arthur Prior in the 1960s. For an introduction to tense logic see [49].
An example would be the temporal statement “Erwin is writing his dissertation”. Although the interpretation, or meaning, of this sentence does not change over time the truth value will most definitely vary as time progresses. In temporal logic the truth value can vary over time.

Temporal logic can be used to reason about how a system behaves over time. There are three variants that have found widespread usage:

2.3.1 Interval Temporal Logic

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time [50].

Since ITL is based on intervals and their sequences, it is possible to describe both concurrent and serial behaviors by specifying behaviors inside intervals and sequences among them. Unfortunately a full set of ITL does not have a decision procedure. However a subset does.

2.3.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is a logic that extends propositional logic with two operators that deal with time. These operators are the unary next and binary until operator. The next operator specifies that a formula will hold in the next state, whereas the until operator expresses that a formula is at least true until the other formula becomes true. From these one can derive operators as eventually and always. LTL deals with the existence of only one possible future path.

An LTL formula can be translated into a deterministic finite state automaton (DFA) [51]. This approach has been used to determine whether or not the LTL formula is satisfiable. The LTL formula is translated into an equivalent DFA
and one verifies if this DFA accepts the empty language. If so, there is no model satisfying the formula. The good news is that the LTL problem is decidable, the negative result is that it is NP complete, this is evident since 3-sat is contained in LTL.

This is reflected in a state explosion that occurs during this translation from an LTL formula to a DFA [52, 53], making it very expensive to establish this correctness.

However various verification tools, such as Maude [54] and Spin [55], are capable of verifying the satisfiability of not overly complex LTL formulas. Although the problem is NP complete the algorithms used have been proven to work well for many practical problems.

2.3.3 Computational Tree Logic

Computational Tree Logic [56] (CTL) can be seen as an extension of LTL. Apart from temporal operators we also introduce unary path operators all and exists. The all operator expresses that for every possible path in the future the formula holds, whereas the exists operator expresses that at least one path exists for which the formula holds.

This extends the notion of time with a set of possible branches over possible futures. Surprisingly the satisfiability problem of a CTL formula is decidable as well [57].
CHAPTER 3
PROBLEM DESCRIPTION

Research groups in both academia and industry have developed prototype systems to demonstrate the benefits of pervasive computing in various application domains. These projects have typically focused on basic system integration—interconnecting sensors, actuators, computers, and other devices in the environment.

Unfortunately, many first-generation pervasive computing systems lack the ability to evolve as new technologies emerge or as an application domain matures. Integrating numerous heterogeneous elements is mostly a manual, ad hoc process. Inserting a new element requires researching its characteristics and operation, determining how to configure and integrate it, and tedious and repeated testing to avoid causing conflicts or indeterminate behavior in the overall system. The environments are also closed, limiting development or extension to the original implementers.

3.1 Non Scalable Integration

Any pervasive system is bound to consist of numerous heterogeneous elements that require integration. Unfortunately, the system integration task itself, which is mostly manual and ad hoc, usually lacks scalability. There's a learning curve associated with every element in terms of first understanding its characteristics and operations and then determining how best to configure and integrate it.

Also, every time you insert a new element into the space, there's the possibility of conflicts or uncertain behavior in the overall system. Thus, tedious, repeated testing is required, which further slows the integration process. Consider a
temperature sensor that needs to be connected to an embedded Java program to periodically report a refrigerated trucks temperature.

3.2 Closed World Assumption

Another problem is that an integrated environment is a relatively closed system; its not particularly open to extensions or expansion, except perhaps accidentally. Its tightly coupled to a combination of technologies that happened to be available at development time. Its thus difficult if not impossible to add new technologies, sensors, and devices after system integration is complete.

An integrated environment is also closed and restricted to only a few participants; the designers, system integrators, and users. Theres no easy way to let a third party participate. For example, an energy- and utility-efficient smart home developed in 2005 might not be compatible with a utility-saving sprinkler system developed in 2006 by a third-party vendor.

3.3 Fixed-Point Concepts

Also problematic is the fact that our experience in building integrated environments is limited by the set of concepts we know at the time of development. This might sound like an always-true statement regardless of whether were doing pervasive, mobile, or distributed computing, but its especially troubling in pervasive computing. Take smart homes, for example. Unlike Nokia phones, you cant upgrade and replace them every six months. Once built for a specific goal (such as to assist the elderly or handicapped, save power, or support proactive health for an entire family), the home will likely be used for decades to come.

Thats why we need to ensure that our smart spaces will be compatible with not-yet-developed concepts. This might not be realistic, but smart pervasive spaces are bound to outlast any known set of pervasive computing concepts. Service gateways and context awareness are two examples of recent concepts that have
steeply influenced how we think of pervasive computing. Surely other new concepts are on the horizon.

3.4 Objectives

The goal of our research is to present a prototype middleware layer and ide capable of remotely browsing current available entities in a smart environment, such as sensors, smart appliances or services, in a smart space, providing information about the capabilities of these entities, offering programming tools and assistance to guide the implementation of new services and avoid restrictions. We will address the issues of enhanced programmability, interoperability and exception capturing.

3.4.1 Enhanced Programmability

Once the components such as sensors, actuators, and smart appliances, are in place, the programmer should be able to program the smart space by pro-actively or reactively specifying predefined behavior under various contexts for these components. The programming model should provide guidance and warnings throughout the development cycle, including programming, debugging and deployment phases. It should provide assistance to allow programmers to focus on programming the behavior of the house, not bothered by technicalities of writing bundles or agents.

3.4.2 Interoperability and Extensibility

It is crucial that the effects of the newly introduced, altered, or eliminated entities or services be reflected in the programming model, and brought to the attention of the system designer or developer. The programming model should guarantee that the introduction of new services or device will not disrupt the already existing devices or services or will alert the developer and resident that the new service will cause a disruption and under which circumstances.
3.4.3 Capability of Exception Capturing

This programming model should be able to capture contradictory instructions or context specifications during the implementation process. The model should be able to capture design flaws and conflicts at both design and implementation stages before being deployed to and activated in the real smart house environment.

3.5 Approach

The most prevalent, current, programming paradigms deal with behavior descriptions. Whether the programs or applications run on a mainframe, PC or embedded computers, this paradigm requires software engineers or programmers to describe how a software artifact behaves by writing the code accepting a given set of input parameters, while the implementation must follow a set of syntactic and semantic constraints specified for a specific programming language. To assist others in understanding what this software does, it is usually a good practice to name this artifact meaningfully such as quickSort or contextInterpreter instead of procedureA.

There are obvious shortfalls with this approach. The naming of these software artifacts is subject to interpretation by the reader, and unless tracing through every single line of code in this artifact, there is no way of knowing whether it actually delivers what its name implies. In short, programmers encapsulate the behaviors into these software artifacts, and bystanders most likely would take their name value as the expected behavior. The implicit nature of these descriptions of behavior becomes especially troublesome in pervasive computing spaces, since they are much more open with almost unlimited potential devices and operations. The fact that smart spaces can actually alter the state of the world is even more unnerving if programmers can only guess what a particular software artifact can and will do by just looking at its name.
Making existing components aware of newly introduced components is a challenge by itself. For existing components to understand the capabilities of these new components is a further challenge. Thus, it is only natural to find the need to describe the effects of the invocation on a particular component. It would then allow reasoning about the behavior of the new component. The description of components should use terms that are commonly understood, in essence. An ontology that is agreed upon is key.

Context driven approach is our attempt to make knowledge explicit. By describing possible contexts in a smart house using standard ontology, the runtime system, as well as any bystanders, can now unambiguously identify which contexts are currently in effect. By defining which actions to take based on what the current contexts are, the knowledge of the behavior of the system is made explicit, in contrast to being implicit in traditional programming. This explicitness is important for several reasons in an open and constant changing system such as a smart space.

First, the entities such as sensors and actuators, both new and existing, behave based on the explicitly defined behavior in context graphs. Thus no guessing is necessary when it comes to cooperation,

Second, since there is no ambiguity in what the current effective contexts are, and all potential actions are explicitly defined, conflicting behaviors and impermissible contexts can be easily checked.

We address the non-scalability and closed world issue by explicitly describing what a device or service is capable of. This description assures that there is no second guessing on the programmers part. If a service or device violates its description it can be detected immediately and appropriate actions can be undertaken,
The separation of this knowledge definition and the availability of the entities in smart houses also greatly enhance its capability to adapt to ever changing nature of open smart spaces hence addressing our third issue of fixed point concepts.

3.5.1 Knowledge Engineering

Knowledge engineering is concerned with the definition of context information, actuators and the relationship between them. During this stage we define how we can derive context from the various sensors. This usually means that we construct a taxonomy and identify how the actuators can influence the various available sensors. This phase is also concerned with identifying which actuator activations are valid under which context. The aim of this phase is to identify potentially dangerous situations and to prevent them by restricting certain operations in a context. For example, we can restrict the activation of a kitchen burner whenever we are not in the kitchen.

Apart from identifying context, we also need to define which behaviors are associated with a context. This involves identifying which behaviors are started and which ones are terminated upon entering and leaving a context. This is also the place where we can install behavior that will be invoked whenever an agent tries to activate an actuator that has been restricted.

3.5.2 Software Engineering

The software engineering phase, on the other hand, is concerned with the definition of behavior. Here we define how various software agents (or services) can interact with each other and how they can activate actuators. The main difference between an actuator and agent is that the latter can never change the physical state of the world; it will need a cooperating actuator to accomplish a state change of the physical world. For example, a DJ agent would organize the electronic music collection of its owner, but will invoke an actuator, such as the stereo, to actually play the music. The interaction with actuators is not guaranteed to succeed. If the
context prohibits the activation of an actuator the application will be notified. The application can now decide to take care of the problem itself, or hand it over to the smart space.

Consider the example of Figure 3–1, if someone is taking a shower and another person in the same apartment wants to flush the toilet. In most cases flushing the toilet will lead to a decrease in the amount of cold water available for the shower, which then in turn can make the shower too hot. In this scenario, flushing the toilet will lead to a situation that is not allowed. The smart space can now invoke a handler that reduces the hot water, preventing the undesired situation.

![Figure 3–1: Negative interaction between entities](image)

In summation, our programming model captures:

- **Knowledge**: This concerns the writing and construction of context information. Construction of context consists of five parts:
  - **Defining conceptual model of the space**: A conceptual model, or taxonomy, is an interpretation of the space in terms of higher level concepts. These concepts capture the possible states of the space by interpreting sensor values. For example we can define that a room can be warm or cold. This conceptual model specifies the users understanding of the space.
  - **Defining desires of a user**: A user in a space has certain desires about the state of the space. These desires are expressed in terms of the conceptual
model. For example a user could prefer a warm room over a cold room during winter, but a cold room over a warm one in summer.

- Defining handlers for invalid activities: A programmer can couple behavior to the invocation of invalid actuators. If the space detects an inconsistent state of the space an appropriate handler should be invoked to either inform the user or to rectify the situation. For example if the space detects that the house is on fire it should inform the authorities.

- Defining actuators and relationships with sensors: The programmer needs to identify which actuators are available in a smart space and specify what kind of relation there exists between the actuators and the sensors. For example a heater has an effect on a heat sensor. Describing this relation allows the space to reason about the effect of activating that actuator.

- **Behavior**: This is the classical software engineering
  - Defining behavior: A programmer implements certain behavior. This is the actual implementation of a service. For example a climate control system that is capable of keeping the room temperate at a certain setting.

  - Associating behaviors with context: A programmer needs to identify which sets of behaviors need to be activated when a context becomes active, in order to satisfy the desires of the user.

3.6 Advantages

The approach that we take has several advantages which we will discuss here.

3.6.1 Explicitness

The biggest advantage of context driven programming model resides in its explicitness. By describing possible contexts in a smart house using description logic, the middleware runtime and bystanders can all unambiguously identify which
contexts are currently active. By defining actions to take based on the currently active contexts, the knowledge on the behavior of system is explicit, contrasting to the encapsulated function calls of traditional programming. This is of great importance since the state of the space is meaningful and essential to its residents.

3.6.2 Interoperability and Extensibility

Entities such as sensors and actuators, both new and existing, behave based on the explicitly defined behavior in the description logic so no guessing is necessary when it comes to collaboration and integration. The explicitness greatly promotes extensibility and interoperability in an open and constantly changing system such as a pervasive space. The separation of the knowledge definition and the availability of the entities in the space greatly enhance its capability to adapt to its changing configurations.

3.6.3 Conflict Detection

Because there is no ambiguity in identifying which contexts are active, and the associated actions to take are explicitly defined, context-driven model is extremely powerful in detecting impermissible contexts and contradictory behaviors. Since contexts of interest are described in the description logic, it is much easier to explicitly identify impermissible contexts. Description logic makes testing for subsumption and membership straightforward, hence making inheritance and overriding a standard part of behavior definition. In addition, atomic contexts along the same dimension are supposed to be disjoint. All these characteristics make it much less possible to create clashing contradictory behaviors.

3.6.4 Capture Environmental Effect

One observation about context-driven model is that it is reactive. Instead of setting a goal and proactively trying to control and coordinate all entities to achieve that goal, systems following this model would passively react to the active contexts observed by executing predefined actions associated with active contexts.
The system has an idea about what the most preferable contexts are, and some about where actions taken may lead to, but it would stay on course until context transitions take place. This reactive nature contributes to its strong capability to capture impermissible contexts and environmental effects.

3.6.5 Automatic Tuning and Machine Learning

The effect of actuators is described in terms of the available context. However it might be possible to bring the effect an actuator has on a sensor by making use of machine learning techniques, thereby rendering the explicit description of an actuator unnecessary.

3.6.6 Scalability

Unlike the service oriented model where each service is encapsulated into a stand alone software artifact, the middleware bundle is the only brain in the context-driven model. The bundle only takes action during context transitions, which are not expected to be frequent once the system has stabilized. The single-bundle reactive approach would fare much better in terms of scalability comparing to dozens of service bundles all proactively collecting, calculating and manipulating at the same time.

Another aspect of scalability arises when we are integrating new services into a space. Due to the explicit description of the service we can determine immediately if there will be a conflict with the existing devices. Their is no need to research the characteristics of the entity since this has been stated explicitly.
CHAPTER 4
OBTAINING INFORMATION FROM SENSORS

A pervasive space needs to obtain information about the physical world. This is done by making use of sensors. Sensors are active objects that provide information about a particular domain, providing data to the system about the current state of the space. A sensor cannot change the state of the space – it can only observe. A temperature sensor, for example, can only observe the current temperature, not influence it. The system is able to interpret the sensor data to generate the context of the space.

A sensor is an autonomous embedded device which has the ability to collaborate with other devices and provides data readings for an application. A sensor is typically battery powered, but this is not a strict requirement. Interaction with a sensor is established by making use of either RS-232, RS-485, ethernet, bluetooth or any other interaction protocol.

A sensor network is comprised of two or more sensor nodes working together to gather information and provide one or more services to an application. The sensor network is simply a logical grouping of sensor nodes. Data and communication between the sensor network gateway and the sensor nodes is transmitted along the sensor network. The sensor network is not a physical medium like RF, or a wire, but rather a logical instrument.

The sensor network developed at the pervasive computing lab integrates various sensors from various sources in a unified way. This takes away the burden of configuring and installing sensors on a sensor to sensor basis.

The sensor platform consists of a hardware part and a software part. The hardware uses a stack based architecture allowing the individual layers to be
redesigned and replaced without affecting any of the other components. This layered architecture allows customization of the sensor nodes for a specific application.

The sensor platform that has been developed contains a software framework that can automatically download the required software used to interact with a specific sensor node from the node itself. This surrogate concept allows the nodes to be easily upgraded without requiring software changes to the sensor gateway. The software can not only reside on the node itself, but also referenced by a URL which the sensor gateway accesses. This referral URL allows the sensor node driver to easily be updated.

4.1 Deriving Context from Sensors Readings

A pervasive space needs to obtain information about the physical world. This is done by making use of sensors. Sensors are active objects that provide information about a particular domain, providing data to the system about the current state of the space. A sensor cannot change the state of the space – it can only observe. A temperature sensor, for example, can only observe the current temperature, not influence it. A smart space typically contains many physical sensors that produce a constant stream of output in various domains. As this data is consumed by the smart space, each physical sensor can be treated as a function the space can call to obtain a value.

**Definition 1 (Sensor).** A sensor is a function that produces an output at a given time in a particular domain.

\[ f_{i}^{j} : U \rightarrow D_{i} \]

Notice that we can have multiple sensors operating in a given domain. Multiple sensors will be indicated by \( f_{i}^{1}, f_{i}^{2}, \) etc. Sensors in the same domain may produce different values due to different locations, malfunction or calibration issues. The smart space system will be responsible for correctly interpreting data.
We will group all available sensors together and define \( f = \prod f_i \) to be a snapshot of all sensors at a point in time.

### 4.2 Using Ontology

Dealing directly with sensor data is rather awkward. Instead we would like to work with higher level information that describes the state of the world. A natural choice for describing knowledge is to use description logics. Using a description logic we can define an ontology that describes the smart space. We will restrict ourselves to taxonomic structures, and therefore will have no need to define roles. Traditionally taxonomies deal with concepts. However we use the term context, since this term is widely used in the pervasive computing community [30, 58].

Since taxonomic structures do not make use of relations they can be considered equivalent to propositional logic. For an in depth discussion on description logics and the relationship between other formalisms we refer to [59]. We will closely follow the definitions given there and introduce the language \( \mathcal{L} \) for context descriptions:

**Definition 2** (Context Descriptions). *Context descriptions in \( \mathcal{L} \) are formed using the following syntax rule:*

\[
    C, D \rightarrow A \quad | \quad (atomic context) \\
    \top \quad | \quad (universal context) \\
    \bot \quad | \quad (non-existing context) \\
    \neg C \quad | \quad (context negation) \\
    C \sqcap D \quad | \quad (intersubsection) \\
    C \sqcup D \quad (union)
\]

*Where \( A \) denotes an atomic context.*
The language described allows us to define the atomic contexts \textit{WARM}, \textit{MODERATE}, \textit{COLD} and \textit{DOOROPEN}. Or more complex context definitions can be constructed, such as \textit{COLD} \cap \textit{DOOROPEN}.

The syntax above tells us how we can construct a valid sentence that described a context. Apart from the syntax we also need to add semantics to the syntax so we can properly interpret the syntactical definition of a context expression. We need an interpretation \( I = (\Delta^I, \cdot^I) \) that contains a non empty domain of interest \( \Delta^I \) and an interpretation function \( \cdot^I \) that allows us to interpret our language. The domain of interest in our case is the possible state of the space, hence \( \Delta^I = f \). The interpretation function maps every concept to a subset of \( \Delta^I \):

\textbf{Definition 3} (Extensional Semantics of \( \mathcal{L} \)). \textit{We define the interpretation \( I \) as follows:}

\[
\begin{align*}
& A^I \subseteq \Delta^I & & \top^I = \Delta^I \\
& \bot^I = \emptyset & & \neg C^I = \Delta^I - C^I \\
& (C \cap D)^I = C^I \cap D^I & & (C \cup D)^I = C^I \cup D^I 
\end{align*}
\]

Using the language \( \mathcal{L} \) we can declare relationships between several contexts. One is the \textit{inclusion} or \textit{is-a} relation. For example, a cat is an animal. The other is the \textit{equivalent} relation. For example a dog is an animal with four legs that barks. Formally these are defined as:

\textbf{Definition 4} (Context Relations). \textit{Inclusion and equality are defined as follows:}

- \textit{Inclusion:} \( C \sqsubseteq D \), interpreted by \( C^I \subseteq D^I \).
- \textit{Equality:} \( C \equiv D \), interpreted by \( C^I = D^I \).

Notice that inclusion \( C \sqsubseteq D \) can be expressed by introducing a new base context \( \bar{C} \) and define \( C \sqsubseteq D \) as \( C \equiv \bar{C} \cap D \). The idea is that \( \bar{C} \) exactly captures the difference between a \( C \) and a \( D \).

The description of how a set of contexts relate to each other is called a \textit{terminology} or T-Box, denoted by \textbf{TBox}. A terminology \textbf{TBox} is a collection of
inclusions and equalities that define how a set of contexts relate to each other. Each item in the collection is unique and acyclic. Contexts that are defined in terms of other contexts are called derived contexts. If we have an interpretation $\mathcal{I}$ that only interprets the atomic contexts, we can interpret the entire taxonomy.

Concepts can now be defined, and the primitive contexts can be given an interpretation. For example: $COLD = \{0,..10\}, MODERATE = \{11,..17\}, WARM = \{18,..25\}$. Notice that this interpretation function is not unique. Different users of the space will most likely have different interpretations of the available concepts.

We can now identify whether the current state of the world is a member of any concept (i.e., to verify whether $u \in f$ satisfies concept $C$ we check that $u^f \in C^f$). Using this we can identify the contexts that are currently active:

**Definition 5** (Active Context:). *The active context of the space $R: U \rightarrow 2^C$ is*

$$R = \{C \mid u^f \in C^f\}$$

Most sensor networks in the literature make use of a hierarchy of interpretation functions, often referred to as context derivation [28, 27, 31]. Since a taxonomy allows us to define inheritance relations we often refer to a taxonomy as a context graph.

Context derivation typically is done by making use of derived interpretation functions. These functions take as input the value (or history of values) from other sensors and produce a new output value. The problem with this approach is that it is no longer possible to verify whether or not the derivation of a particular context is sensible or not, that is there is an actual state of the world that satisfies the defined context. It is straightforward to construct a sensor network that derives inconsistent information. The strength of a description logic lies in the fact that every context that we define is sensible and can actually occur in a space.
4.3 Temporal Context

Interpreting the state of the world only gives us an understanding of the world at a single moment in time. By only looking at a single moment of time it becomes impossible to describe events or contexts that depend on some temporal ordering of events. Using merely snapshots we cannot describe an activity of daily living. For example the activity of doing laundry consists of a sequence of events. First we gather the laundry, we obtain detergent and we activate the washing machine.

Another scenario is a person walking indoors. We can detect this sequence of events by making use of two pressure sensors under the doormat as shown in figure 4–1. One sensor on a doormat that is outdoors and one that is indoors. As soon as a person comes inside he will press the outdoor sensor followed by the indoor sensor. If a person leaves the house it will happen in reverse order. First the indoor sensor is pressed then the outdoor sensor is pressed. Figure 4–2 show this sequence is a diagram.

To express this context we somehow need to incorporate time into our taxonomy. This can be done by extending the language $\mathcal{L}$ to handle time.
4.3.1 Syntax of Temporal Context

We introduce, an extension to our taxonomy $L$ that allows to express and reason about contexts over time. We model time as a discrete change in the state of the space and follow the so called external approach \[60\]. In the external approach the very same individual can have different “snapshots” in different moments of time that describe the various states of the individual at these times. In our case the individual is that state of the pervasive space.

In this approach context definitions remain invariant, meaning that the meaning of a concept does not change over time. The concept $COLD$ for example will have the same interpretation regardless of the current time. The membership of an individual can however change over time. In moment in time the house can be $COLD$ and another moment in time it is not. The same individual is classified differently based upon time.

We introduce the language $LT$, by extending $L$ with time constructs. The language $LT$ is used to describe temporal contexts. We follow the approach taken by \[61\], which in turn is based on tense logic \[62, 63\].

Definition 6 (Syntax of \(\mathcal{L}T\)). Context descriptions in \(\mathcal{L}T\) extend \(\mathcal{L}\) by adding the following syntax rules:

\[
C, D \rightarrow \ominus C \quad | \quad (\text{previous instant})
\]

\[
\ominus C \quad | \quad (\text{next instant})
\]

\[
C \ U D \quad | \quad (\text{until})
\]

\[
C \ S D \quad (\text{since})
\]

The addition of these four constructs are quite powerful. For example we are able to introduce the following abbreviations with regards to time:

\[
\triangleright C \equiv \top \ U C \\
\triangleright C \equiv \ominus \ominus C
\]

\[
\triangleleft C \equiv \top \ S C \\
\triangleleft C \equiv \ominus \ominus C
\]

Where \(\triangleleft\) reads as “once”, \(\triangleright\) as “eventually”, \(\ominus\) as “always” and \(\ominus\) reads as “it has always been.” The constructs allow us to express things as \(\text{OwnerAtDoor} \ominus \ominus \text{Dooropen}\), the house will open the door in the next instant if it detects the owner at the front of the door.

We can further enhance our expressiveness by adding the following fixpoint abbreviations as well:

\[
C \text{ before } D \equiv \ominus \ominus \neg D \wedge \ominus (C \lor C \text{ before } D)
\]

Using these abbreviations we can describe the example given in figure 4–1, and its opposite as follows:

\[
\text{WalkIndoors} \equiv \text{OutdoorMat} \text{ before } \text{IndoorMat}
\]

\[
\text{WalkOutdoors} \equiv \text{IndoorMat} \text{ before } \text{OutdoorMat}
\]
### 4.3.2 Semantics of Temporal Context

In order to reason over time we need a temporal structure $T$ that captures our notion of time. For our purpose we define $T = (\mathcal{P}, <)$ where $\mathcal{P}$ is a set of time points with a strict partial order $<$. We assume that there is a start of time, which we denote with $t_0$, which has the property that $\forall t \in \mathcal{P} \langle t_0 < t \lor t_0 = t \rangle$.

The interpretation of $L_T$ is a triple $I = \langle T, \Delta^I, \cdot^I \rangle$, where $T = (\mathcal{P}, <)$ is the time domain, $\Delta^I$ the individual domain and $\cdot^I$ is the interpretation function.

The points in $T$ form the infinite informal time scale. A state is basically a point in time. Notice that we nowhere describe “how much” time it takes to go from one state to the next state. Depending on the required precision of the observations we can define the amount of “time” it takes to make a transition. The semantics can now be given as follows:

**Definition 7** (Semantics of $L_T$). The semantics of $L_T$ are as follows:

\[
(\oplus C)^T_t = \{ a \in \Delta^T \mid \exists t'. t < t' \land a \in C^T_{t'}, \neg \exists t''. t < t'' < t' \} \\
(\oplus C)^T_t = \{ a \in \Delta^T \mid t > t_0 \land \exists t'. t' < t \land a \in C^T_{t'} \land \neg \exists t''. t'' < t' < t \} \\
(C \ U D)^T_t = \{ a \in \Delta^T \mid \exists t'. t < t' \land a \in D^T_{t'} \land \forall t''. t < t'' < t' \rightarrow a \in C^T_{t''} \} \\
(C \ S D)^T_t = \{ a \in \Delta^T \mid \exists t'. t < t' \land a \in D^T_{t'} \land \forall t''. t' < t'' < t \rightarrow a \in C^T_{t''} \} 
\]

The semantics capture our notion that the context definitions remain invariant over time. The membership assertions of individuals however can change over time.

#### 4.3.3 Describing Events using Temporal Context

The language $L_T$ allows us to reason how the state of the space behaves over time. There are however some counter intuitive observation we can make with regards to the logic. Suppose we have the sequence as given in figure 4–3 where $T$ denotes that the state of the world satisfies the given context.

As we can see the context in the house is active before we actually entered the house, even worse since we obtain our snapshots one by one we never enter the
context *WalkIndoors*, since this context is true in the past. The context before should be active after time interval 7, not before!

The problem stems from the usage of future tense instead of past tense in our definition of before. In the case of recognizing context we would like our expressions to be true as soon as we have recognized a sequence of context states. Meaning that an activity of daily living should be described in terms of the past not of the future. Replacing the description using past tense operators would solve this issue.

Another problem with the definition of before is that it is a so called *weak* operator. A \( \text{before} \ X \) does not necessarily imply that \( X \) will be true at a certain point. Again, this is not desirable.

The easiest solution to these problems is to give direct semantics to the before operator, instead of expressing them in terms of others. The semantics of before can be given as follows:

\[
(C \text{ before } D)_{i}^{T} = \{a \in \Delta^{T} \mid \exists t'. t' < t \land a \in C_{i}^{T} \land \exists t''. t' < t'' < t \rightarrow a \in D_{i}^{T}\}
\]

### 4.3.4 Computational Complexity of Temporal Context

Burgess gives a complexity analysis of \( \mathcal{ALCT} \) [61]. The language of \( \mathcal{ALCT} \) is similar to the language we described above, except with the addition of roles and removal of temporal operators that deal with the past. The addition of past
operators does not make the language any more powerful, the addition of roles however does.

He proves that concept satisfiability in $\mathcal{ALCT}$ has the same time complexity as $\mathcal{ALC}$ if we take $\mathcal{P}$ to be a discrete time structure like the natural numbers. Thus reasoning over $\mathcal{ALCT}$ is PSPACE complete. Unfortunately the time bounds for tense logic transfer to this class and hence the problem is EXPTIME hard [64].

However implementations of reasoning engines that can deal with temporal description logics exist [65, 66] and despite the computational complexity of these algorithms, they perform reasonably well for “normal” applications. Notice that OWL-DL has similar complexity results but is still widely used in an effective manner. For example [67] shows how OWL can be successfully used in a pervasive space.
CHAPTER 5
DESCRIBING ACTUATORS AND SERVICES

The devices in a smart space are not turned on randomly. We turn on an actuator because of its effect. Every actuator has a so called *intentional* effect on the world. The intention of turning on the heater for example is to raise the temperature.

By describing the intentional effect of an actuator we can use this to reason about the effect of the invocation of a certain actuator. This can assist the developer when programming the space. We can identify for example that invoking the A/C and the heater at the same time is rather strange.

On a higher level we have services that accomplish certain goals. For example a temperature control service might be able to keep the temperature at a constant. This service can accomplish this by making use of the intentional effect of the available actuators.

If we have a description of how an actuator or service behaves over time, we can use this to prevent conflicts and assist a developer in how to accomplish a state in the space.

The ontology $\mathcal{LT}$ can now be used to describe the effect of the invocation of a service or actuator. The description can be used by the space to reason about the effect of the activation of a particular actuator or service. Consider the following
description of a heater and actuator:

\[ \text{Heater} \land \text{Moderate} \rightarrow \text{Moderate} \cup \text{Warm} \]
\[ \text{Heater} \land \text{Cold} \rightarrow \text{Cold} \cup \text{Moderate} \]
\[ \text{Airco} \land \text{Moderate} \rightarrow \text{Moderate} \cup \text{Cold} \]
\[ \text{Airco} \land \text{Warm} \rightarrow \text{Warm} \cup \text{Moderate} \]

If the current state of the world is moderate and we invoke both the heater and the air-conditioner we can detect a logical inconsistency. 

\((\text{Moderate} \cup \text{Cold}) \sqcap (\text{Moderate} \cup \text{Warm})\) are logically inconsistent, there is no existing model that satisfies this formula. We detect that the invocation of these two actuators lead to an unpredictable state, or will result in an ill defined state.

Similarly we can describe services in terms of how they effect the space. For example a climate control service can be described as △ ⊞ Moderate.

After activating the climate control there will be a time after which it is always moderate. In general:

**Theorem 1.** Let \( p \) and \( q \) be the descriptions of actuator (or service) \( a_i \) and \( a_j \). \( a_i \) and \( a_j \) are said to be in conflict in state \( r \) if \( p \sqcap q \equiv \bot \).

**Proof.** This is trivial. If \( p \sqcap q \equiv \bot \) then there doesn’t exist a model that satisfies \( p \sqcap q \). This directly implies that there is no sequence of states that satisfies these formulas. \( \square \)
CHAPTER 6
PROGRAMMING THE SPACE

So far we have described a space consisting of sensors and actuators. The key element that is missing is the user. To model the user we roughly follow the belief-desire-intention [68] model, in the sense that the user observes the state of the space and, based upon this information, commits to a plan of action. We have the following ingredients:

- **Desires:** A user has a set of preferences about the state of the world. These preferences dictate how the world should be at that given moment. For example, when a user is going to bed, he or she would like the doors locked.

- **Belief:** This is the current observed state of the world, interpreted by a taxonomy. In any given state of the world the user has a set of preferences about the world. When the state changes, the set of preferences can change as well.

- **Intention:** This is the plan to which the user commits. In a smart space this would be the activation of a set of actuators. The aim of the activation is to fulfill the desire mentioned before.

The idea is that the user observes that state of the world. The user then classifies the state according to the users specific taxonomy. This classification gives rise to a set of contexts that are active. Associated which each context is a set of behaviors. The intention of these behaviors is to change the current state to a more desired state. For example, if user interprets the world as “dark” we turn on a lamp with the intention to reach the desired state of “light.”
6.1 Beliefs-Desires-Intentions of Users

As we mentioned earlier a user has a belief about the current state of the world. The current belief about the state of the world can be captured with a taxonomy and an accompanied interpretation. The interpretation maps the values obtained from a sensor to the atomic concepts. We express the intentions of the users by a sequence of actions the user wishes to execute in a particular context. Formally:

**Definition 8** (User). A user is a 3-tuple

\[ B ::= (T, \mathcal{I}, I) \]

Where \( T \) and \( \mathcal{I} \) are the taxonomy accompanied by the interpretation of the user. Let \( I : C \rightarrow S \) be a mapping from context to statements.

We can now define a space consisting of sensors and actuators and a user using the following operational semantics:

**Definition 9** (Statement Sequence). Given a set of actuators \( A \), a sequence of actions \( S \) is defined as follows:

\[ S ::= \uparrow a_i | \downarrow a_i | S_1; S_2 \]

Where \( \uparrow a_i \) turns the actuator \( a_i \in A \) on, \( \downarrow a_i \) turns the actuator \( a_i \) off. With \( ; \) we denote action prefixing. We first perform action \( S_1 \) after which we execute the remaining statements \( S_2 \).

**Definition 10** (Programmable Pervasive Space). A programmable pervasive space is a tuple consisting of

\[ P ::= (B, U, S, 2^A) \]

Where \( B \) is the representation of the user, \( U \) the state of the space as observed through our sensors, \( S \) the statements we are currently processing and \( 2^A \) the set of actuators that are currently active.
A space changes over time due to nature. We model the effect of nature by a transition that changes the state of the space, and observe the changes of the space through our sensors.

- **Nature:** \( \langle b, S, f(u), a \rangle \rightarrow \langle b, S, f(u'), a \rangle \) where \( f(u) \neq f(u') \)

The other way in which the space can change is by actuators that the space controls. The following transitions capture the turning on and off of actuators:

- **Activation:** \( \langle b, \uparrow a_i, u, a \rangle \rightarrow \langle b, \epsilon, u, a \cup a_i \rangle \)
- **Deactivation:** \( \langle b, \downarrow a_i, u, a \rangle \rightarrow \langle b, \epsilon, u, a \setminus a_i \rangle \)

If the set of active contexts has changed, we obtain the intentions from the user and execute the desired sequence of actions:

- **Context Change:** \( \langle b, f(u), S, a \rangle \rightarrow \langle b, f(u'), i(R(f(u'))), a \rangle \)
  
  whenever \( R(f(u')) \neq R(f(u)) \)

### 6.2 Inheritance and Override

The concept of subsumption introduces inheritance.

**Definition 11** (Inheritance).

\[
C \subseteq R \land C \sqsubseteq D \land \uparrow a_i \in I(D) \rightarrow \uparrow a_i \in I(C)
\]

*Naturally this holds for turning off:*

\[
C \subseteq R \land C \sqsubseteq D \land \downarrow a_i \in I(D) \rightarrow \downarrow a_i \in I(C)
\]

### 6.3 Scenario and Programming Procedures

We have formally established a context-driven programming model for pervasive spaces, employing definitions for sensors, actuators, users, and contexts. But how exactly would this model help programmers create or modify smart spaces? In this section we present a scenario and outline the programming procedure to demonstrate the feasibility of the context-driven pervasive space programming model.
6.3.1 Lightening Control Application

The following scenario is set in a smart apartment at a retirement community. It is a studio apartment intended for a single occupant, and the service to be programmed is the ambience lighting control.

The ambience lighting inside the apartment is automatically adjusted based upon the active contexts. As a general rule, when the room is too dark, lamps and other light fixtures should be turned on; when the room is too bright, blinds should be closed to maintain a comfortable living environment. When the resident is sleeping, the apartment should be kept dark. When the resident is watching TV, in order to ensure the best viewing experience, the ambience lighting should be kept at a moderate level.

6.3.2 Describing Context

The first step in programming the described lighting control in this smart apartment is to properly model the physical aspects of the space. At any moment, we can describe this smart space as being in certain states; for instance, the smart space can be described as Bright, Dark, ResidentwatchingTV, or Residentsleeping.

While human beings can take a glimpse at the apartment and almost immediately identify these contexts a smart space to make sense of all these, two criteria must be satisfied. First, there must be appropriate sensors that can perform the relevant measurements. For instance, in order to identify Bright or Dark, some sort of luminance sensor must be present in the smart space. And, without a thermometer, there is no way for a smart space to observe whether the room temperature is Hot or Cold. Second, there must be an interpretation of the readings generated by the sensors to more abstract contexts. For example, we can interpret any reading of higher than 600 lux from a luminance sensor as Bright, readings under 400 lux as Dark, and anything in between as Illuminated.
In addition to these atomic contexts that can be directly measured by physical sensors, it is also useful to have derived contexts, which can be defined with the help of taxonomy operations. For the ambience lighting control service, the derived contexts such as *BrightTVOn*, *IlluminatedTVOn* and *DarkTVOn*, which are built from atomic contexts *Bright*, *Illuminated*, *Dark* and *TVOn*, are extremely helpful in identifying the current context of interest when trying to decide the next appropriate action the system should perform.

6.3.3 Associating Behavior

Once contexts of interest (both atomic contexts and derived contexts) are defined and interpretation functions are devised, the smart space can now understand which contexts are currently active. For each reading of each sensor with properly defined contexts in its associated domain, that reading should be able to be interpreted as exactly one active atomic context in that domain. For instance, if the luminance sensor were to report a reading value of 700 lux, atomic context *Bright* would be recognized as active, while a query of a simple sensor in the TV set should report either *TVOn* or *TVOff* is the current active atomic context. These active atomic contexts can be used to generate further active derived contexts, when all its child contexts are active. For instance, *BrightTVOn* would become active automatically when the luminance sensor reports active *Bright* context and TV set reports active *TVOn* status.

In order to achieve the goal of automatically adjusting ambient lighting, we have to find some way for the smart space to interact and influence the real world. This can be achieved by manipulating actuators. With the current version of our context-driven programming model, actuators are only allowed binary operations, which means that they are either turned on or off. Each actuator that is turned on should generate at least one intentional effect, which results in a context transitions. For example, turning on lamps would brighten the room (that
is, the space would transition between contexts, such as \textit{Dark} $\rightarrow$ \textit{Illuminated} or \textit{Dark} $\rightarrow$ \textit{Bright}, while closing the blinds would darken the room (\textit{Bright} $\rightarrow$ \textit{Dark} or \textit{Illuminated} $\rightarrow$ \textit{Dark}).

How does the smart house know when to turn on the lamps or close the blinds? That is where users come in. The occupant of the apartment should have the ultimate power to make decisions. The resident calls the shots on defining what \textit{Dark} or \textit{Bright} mean, and the context graph should be built based on his or her beliefs so as to help the user truly understand the current active state of the entire smart space, as well as facilitate the planning of actions. The user also gets to express his or her desires by specifying the preferable contexts. This ultimately defines the goals the smart space should work towards when deciding which actions (if any) to take. In essence, the space must find and execute a route from the current active contexts to the promised preferable contexts using the only tools it has to change the state of the space: actuators. And since the smart space knows about the current active contexts, the final preferable contexts, and the intentional effects of each actuator, it has all the information needed to devise this route and fulfill the user’s intention.

6.3.4 Connecting Sensors

We now return to the scenario of programming ambience lighting control. In order to provide this service as described, three different sensors have to be present in the apartment: a luminance sensor, a TV set sensor, and a smart bed that can detect whether the resident is on it.

The TV set sensor and smart bed generate Boolean values as to whether the TV set is on or off, and whether the resident is in bed or not. The luminance sensor, however, requires an interpretation function between numerical reading values and atomic contexts \textit{Bright}, \textit{Illuminated} and \textit{Dark} (the values for these contexts have been defined above).
In addition to the sensors, we assume two actuators are available in the apartment: one to control the lamps and one to control the blinds. The intentional effects of each actuator are also described above.

As for contexts, there are atomic contexts such as *Bright*, *TV On* or *InBed* that can be interpreted directly from sensor readings, and there are also derived contexts such as *DarkInBed* and *IlluminatedTV On*.

Since the user would prefer the lights to be turned off completely while he or she sleeps, but prefers adequate lighting while watching TV, *DarkInBed* and *IlluminatedTV On* are potential goals (depending on the user’s current intentions), and are therefore preferred contexts.

For each non-preferred context, an associated statement (consisting of a sequence of actions manipulating actuators) is attached, which specifies how to move from that context to another. Hence the smart space identifies *DarkInBed* and *IlluminatedTV On* as preferred contexts, and will strive to make either one of them the active context by executing the action statement associated with the current active context. Figure 6.3.4 shows the context graph described in this section.

The derived contexts automatically inherit the action statements from their child contexts. Hence if *BrightInBed* is associated with action closing the blinds then *BrightTV OnInBed* should, unless specified otherwise, also retain the same action. However, an interesting scenario occurs when we consider a situation with the resident lying in bed while watching TV before going to sleep in the currently illuminated room.

In this situation we have three active derived contexts: *IlluminatedTV On*, *IlluminatedInBed*, and *IlluminatedTV OnInBed*, with the last one derived from the previous two. We observe that there are conflicting inherited action statements. To resolve these conflicting actions, with one trying to close the blinds while the
other indicating that no action is needed, we need to define a new action statement
when IlluminatedTVOnInBed is active. Since the resident is still watching TV,
the lighting should be kept illuminated until she or he decides to go to sleep.

The context driven programming model is effective in detecting conflicting
actions and can prompt the user for resolution. We also observe how inheritance
and overriding action statements enable the development of robust smart spaces.
CHAPTER 7
INTEGRATED DEVELOPMENT ENVIRONMENT

With the context driven programming model formally defined, we are now ready to explore how programmers can actually program a smart space.

7.1 Design of the Context Graph

The programming procedure would start with designing a context graph, which is a semantic web that is composed of interesting states of the space. There are two types of contexts in a context graph, the atomic contexts and derived contexts. Atomic contexts differ from derived contexts in that they can be directly associated with readings of one particular type of sensor, while derived contexts are composed of multiple atomic contexts. For example, cold and hot are two atomic contexts, because they can be associated with thermometers, with a predefined range of readings from the thermometer. On the other hand, contexts “cold and dry” “hot and dry” derived contexts, since they cannot be directly associated with sensors, but instead can only be derived from a combination of atomic contexts such as cold and dry for “cold and dry” context.

The design of a context graph can proceed without knowledge of availability of various entities in the house, and can be applied to different smart spaces. However, for each smart space, the contexts of interest would vary based on the collections of available sensors and actuators. The contexts of interest are also affected by such things as the residents activities and preferences and the caregivers primary concerns.

It would be natural, therefore, to assume one possible practice when designing the context graph. This would be to get a generic context graph as a template, and then customize for each smart house and resident based on factors such as
availability of sensors and actuators, or residents preference. There would be no problem should the context graph contain either more or fewer contexts than could be sensed by available sensors, although if the context graph cannot interpret reading from some sensors, it would render these sensors useless. This property allows a generic context graph template to be used, and also provides system-wide robustness when some entities fail or new ones are introduced to the smart house.

7.2 Interpretation of Sensor Readings

To allow the runtime platform to interpret sensor readings, the entire range of possible readings from each sensor must be mapped to disjoint atomic contexts. For example, for a thermometer which is capable of reading temperatures between $-20^{\circ}\text{F} \sim 222^{\circ}\text{F}$, we need to specify that $20^{\circ}\text{F} \sim 55^{\circ}\text{F}$ would map to cold atomic context, and $55^{\circ}\text{F} \sim 70^{\circ}\text{F}$ would map to cool atomic context, and $70^{\circ}\text{F} \sim 80^{\circ}\text{F}$ would map to warm atomic context, and $80^{\circ}\text{F} \sim 222^{\circ}\text{F}$ would map to hot atomic context. The association of a range of possible reading values in integer or real numbers, Boolean values or enumerated strings to certain defined atomic context enables the runtime platform to make sense of the sensor readings as the current state of the smart house, which leads to the decision of which actions to take.

7.3 Desired Behaviors under Each Context

Contexts defined in a context graph represent the whole spectrum of possible interested states that can be used to describe the current standing of the smart house. There are contexts that are not allowed and should be avoided, if, unfortunately, the current context falls into one of these impermissible contexts, some sort of emergency actions should be taken to move out of this context as soon as possible. For example, the context of “extremely hot and smoking” would be an example of impermissible context, and extreme measures such as call 911 and initiate fire depressant would be warranted. There are others that are situations considered to be normal, but not quite the most desirable situation. In
those contexts, actions should be specified so the state of the smart house would gradually move toward more desirable contexts. The specified actions in order produce to this movement would be called desired behavior associated with the context. For example, in a cold February in a northern climate, it is quite possible the current context within the smart house would be “cold and dry”, which is normal but not quite desirable. The more comfortable, hence desirable context in this example would be “warm with moderate humidity”, and the desired behavior associated with “cold and dry” context would therefore probably be turn on the heater and the humidifier. The system designers and programmers would program smart houses by associating behaviors to each of the specified contexts in the context graph. This would normally lead to context transitions that move closer to the desirable contexts.

7.4 Mapping Physical Sensors and Actuators to the Context Model

After the completion of the previous three steps, we now have the knowledge to describe all contexts of interest of the house, the knowledge to interpret the sensor readings and decide what the current contexts are, and the knowledge of what actions to take to steer the house toward the desirable context. But how do we apply all this knowledge to the physical entities in the smart house? An array of disjointed atomic contexts like cold, cool, warm and hot is theoretically great in describing the context of the house, but would not be effective unless certain sensor(s) actually available in the house are associated with these contexts. The same would apply to the actuators, which need to be manipulated for actions associated with each context. This association is established at runtime, when the runtime platform dynamically links the contexts and actions with existent sensors and actuators based on their associated measuring dimensions, locations, and other information. With the programmed knowledge installed and physical entities linked, the smart house is now ready to operate based on programmers directives.
7.5 Implementation of the IDE

In this section we look at how we implemented the development environment.

7.5.1 Premise and Platform Selection

The existing service platform for GTSH is built upon OSGi. It provides excellent facilities to manage the lifecycle of all services and entities, implemented as individual bundles, and handles dynamic addition or removal automatically. It also provides good APIs such as wiring between data sources and sinks, and service registration to allow dynamically fulfilling bundle dependencies. OSGi Alliance, which maintains the technology, consists of many major players in the computer and appliance industry, assuring the wide acceptance of OSGi. Realizing that although providing an IDE for smart spaces and exploring reasonable programming models is relatively novel for pervasive computing researches, the effort of programming bundles in Java or establishing context graphs with ontology are not. Hence we decided early on to implement this IDE as an enhancement or attachable plug-ins to established development tools instead of a full fledged standalone software. Two potential targets are identified: (1) Eclipse which is one of the most predominant IDE used by Java developers, allowing easy bundle authoring, and (2) Protégé which is one of the most heralded tool for ontology and semantic web editing. After careful consideration, we decided that Protégé was preferable for the initial phase of our work since most of the programming effort using the context driven programming model requires establishing and editing the semantic web and context graph rather than implementing Java code. Protégé also has the capability to employ reasoning engine which can establish whether a concept subsumes, is equivalent to another or whether a certain individual is a member of a particular concept, which is crucial in exception capturing and active contexts derivation in the context driven programming model. Should we have a future need to incorporate heavy bundle coding in Java, the current enhancement
to the Protégé would be carried over to the Protégé plug-in for Eclipse to preserve the current effort and provide seamless integration between the two environments.

7.5.2 Requirement Analysis

Small scale interviews regarding the requirements of the IDE were conducted among the in-house investigators who participated in the initial setup of GTSH. With their hands-on experience in actually programming a full-scale smart house, the following requirements for the IDE were identified:

1. capability for browsing existing sensors, actuators and services in the smart environment.
2. capability for browsing the context compositions.
3. capability for viewing and editing the interpretation of sensor readings to associated atomic contexts.
4. capability for viewing and editing the behaviors associated with each context.
5. capability for creating contradictory behaviors and impermissible context checks.
6. capability for installing and deploying the programs to the runtime platform.
7. Ease of use to allow domain experts to modify current or create new behaviors of smart houses.

7.5.3 Enabling Middleware Runtime

As part of the GTSH project, middleware architecture has been proposed and developed [15]. The middleware serves as the backbone of smart house, allowing physical entities such as sensors and actuators to be automatically integrated into the environment upon entry. The middleware also empowers designers to define contexts and knowledge and program the behavior of the smart house.

There are two software artifacts that are essential in actual enabling the programming process of smart houses. There is, of course, the IDE that provides tools and guidance that allow system designers to actually implement code and/or
configure and specify knowledge that depicts the expected behavior of the smart environment. There is the middleware runtime that would likely reside somewhere in the smart house that actually interprets and executes these instructions created using IDE. The interactions between these two software artifacts are shown in Figure 7–1.

The current active middleware runtime is implemented on an OSGi platform as a bundle. This bundle serves as the brain of the smart house. It is connected to various bundles representing sensors via OSGi wiring API, and feeds these proactively collected sensor readings into atomic context interpreters. These classified atomic contexts are fed into a Jena and reasoning engine to activate all the active contexts. The middleware bundle then looks up the behavior associated with these contexts and triggers the programmed actions, mostly by either turning on or off certain related actuators. Current version of this middleware also performs a predefined association between physical sensors or actuators and the entities represented in the OWL file. As the current project of enhancing the context driven programming model and implementation of IDE progress, there
also have been minor revisions of the middleware bundle to support these changes and enhancement. In the wake of the process of developing the IDE, there are undoubtedly iterations among IDE, the middleware bundle and the programming model. Since they actually represent different facets of the same thing, they are definitely tightly coupled. System designers use IDE to program and configure the smart house, while the programmed code and configurations are interpreted and executed by the middleware runtime. The programming model provides the theoretical background and design rational for both the IDE and the middleware.

7.6 Components of the IDE

In this section we look at how the various components in the IDE help the developer programming the pervasive space.

7.6.1 Entity Definition View

As shown in figure 7-2, this view provides the tools for browsing sensor and actuator entities currently available in the smart house. It also provides the graphical interface for editing the sensor readings mapping to atomic contexts, as well as specification of the effects of actuators in terms of switching between atomic contexts. This view is implemented as a plug-in tab to Protégé 3.1, and the visualizations are implemented using standard Java Swing component. All information and specifications edited and displayed on this view are stored in the OWL file that serves as the blueprint of the smart house under development. It short, the purpose of this view is to provide the system designer with the knowledge about the availability of various sensor and actuator entities, and detailed information about them, such as location, entity ID, the type of measurement., This allows system designers to program and update certain aspects of each entity presented, such as readings value mapping of sensors and default effects of actuators.
7.6.2 Behavior Definition View

As shown in Figure 7–3, this view provides the tools for browsing the context graph of the smart house, editing the graph to reflect the contexts of interest, and defining expected behavior associated with each context of interest. It also provides visualization of the potential migration path of contexts as the effect of programmed behaviors that should be invaluable in guiding the programming process. Instead of asking system designers to draw the entire context graph, a list of various atomic contexts are provided and system designers can simply pick the derived context of interest, and the IDE would automatically derive the context graph based on ontology and the inherit relations between contexts. This view is implemented as another plug-in tab to Protégé 3.1, and the visualizations are implemented using standard Java Swing component. All information and specifications edited and displayed on this view are stored in the OWL file that serves as the blueprint of the smart house under development. In short, the purpose of this view is to help system designers grasp the programmed behaviors and their effects, and modify these behaviors if so desired.
7.6.3 Impermissible Context Checker

This checker serves as a safeguard mechanism before a program is deployed to the real remote smart houses. Some of the pre-deployment checking includes overlapping of sensor readings for disjoint atomic contexts, contradictory actions specification associated with same context. This checker is implemented using Jena reason engine, which automatically examines the OWL file programmed using two views described in the previous sections. Should any impermissible context be found, the IDE prevents the deployment to avoid destabilizing the target smart house.

7.6.4 Communication Module

This module enables communication between the IDE and the middleware runtime. The communications should happen only twice during the programming process, once at the beginning when retrieving a snapshot of current running
programs and available sensor/actuator entities information from the smart house. System designers would program the smart house based on this retrieved snapshot. At the end of the programming process, after the context checker verified the design, the OWL file is uploaded to the target smart house and replaces the current running program. This minimal communication pattern is chosen for two reasons.

First, since smart spaces are an open and volatile environment, it is assumed that sensors, actuators or smart appliances can be brought into the house or taken out at any time. With the large number of these entities, failure is could be the norm. Hence if the IDE constantly tracks all the changes and reflects them on various views in real time, it would cause confusion and inconvenience to designers at work.

Second, since the IDE is used for programming, not monitoring, continuous real time communication does not seem necessary. Instead, we opt for pre-deployment checking to see if any changes in the smart house would merit further changes of the target OWL file. This compromise would grant a relative stable working environment for designers during programming process, yet maintain integrity of the program while using minimal communication bandwidth. All the necessary knowledge such as sensor reading mappings, actuator effects and context associated behaviors needed to specify how smart house should behave are all structured as part of the target OWL file.

The retrieval and deployment of information between IDE and middleware runtime is made extremely simple as corresponding to downloading and uploading the OWL file respectively. The communication module is implemented as a servlet. System designers can select which smart house to connect to by specifying its IP or domain name, and separate “connect” and “deploy” button would initiate either download or upload sequence respectively. In short, this module provides the bridge between IDE and middleware runtime located at a remote smart house.
Figure 7–4 shows the snapshot during context checking and program deployment process.

Figure 7–4: Impermissible context verification
DESCRIPTION LOGICS

Since the framework presented in this dissertation depends heavily on description logics we will delve into more details and provide the mathematical foundation of a description logic. A good understanding of the foundations will show that the interpretation of sensor data using a description logic is straightforward.

Description logics are concerned with describing knowledge about a problem domain. A knowledge base consists of two parts:

- **Intensional knowledge (T-Box) TBox:** The intensional knowledge contains the general knowledge about a problem domain. It is defined in terms of a terminology. A terminology defines a set of relations between various concepts. For example an *elder* is a person. An *elder* is at least 75 years old.

- **Extensional knowledge (A-Box) ABox:** Describes the specific knowledge to a particular domain. A world description defines a set of individual instances belonging to the defined terminology. For example: *Mathilda is 85 years old.*

The knowledge base can now be used to derive additional information that is not explicitly stated in the extensional knowledge. For example we can derive that *Mathilda* is an *elder*, since she is 85 years old and anyone who is older than 75 is an elder.

### A.1 Describing Intensional Knowledge

The aim of intensional knowledge is to describe relationships between various concepts in a problem domain. The key ingredients are concepts. Concepts are names that refer to concepts that are used in the problem domain. There are two types of concepts:
• **Primitive:** A primitive concept (denoted by the letters $AC$) is a concept that stands on itself. It refers to a valid concept in a problem domain. For example the concept $Location$ could refer to a location in the space.

• **Derived:** A derived concept is a concept (denoted by the letter $C$ or $D$) that is defined in terms of relations to a set of primitive concepts. For example we could say that a $Room$ has a $Location$ inside a $House$.

Apart from simple names to organize concepts we can also add properties to concepts. Properties are binary relations associating individuals from different concepts together. For example a person can play the role of a parent, associating another person to this person. We will use the letter $R$ to denote a role.

We can now give the syntax of the language as follows:

**Definition 12** (Basic language $\mathcal{ALC}$ [59]). *Concept descriptions in $\mathcal{L}$ are formed using the following syntax rule:*

\[
C, D \rightarrow A \quad | \quad (\text{atomic concept}) \\
\top \quad | \quad (\text{universal concept}) \\
\bot \quad | \quad (\text{bottom concept}) \\
\neg C \quad | \quad (\text{concept negation}) \\
C \cap D \quad | \quad (\text{intersection}) \\
C \cup D \quad | \quad (\text{union}) \\
\forall R.C \quad | \quad (\text{universal quantifier}) \\
\exists R.C \quad (\text{existential quantifier})
\]

The language $\mathcal{ALC}$ allows us to express various concepts. For example we could define a “Happy professor” using the atomic concepts $Tenure$ and the role $Funding$:

\[HappyProfessor = Tenure \cap (\exists \text{Funding}.\top)\]
A professor is happy if he (or she) has some form of funding and tenure.

The syntax above tells us how we can construct a valid sentence that described a concept. Apart from the syntax we also need to add semantics to the syntax so we can properly interpret the syntactical definition of a concept expression. We need an interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ that contains a non empty domain of interest $\Delta^\mathcal{I}$ and an interpretation function $\cdot^\mathcal{I}$ that allows us to interpret our language. The interpretation function maps every concept to a subset of $\Delta^\mathcal{I}$ and every role to a subset of $\Delta^\mathcal{I} \times \Delta^\mathcal{I}$:

**Definition 13** (Extensional Semantics of $\mathcal{ALC}$). We define the interpretation $\mathcal{I}$ as follows:

- $\mathcal{A}^\mathcal{I} \subseteq \Delta^\mathcal{I}$
- $\top^\mathcal{I} = \Delta^\mathcal{I}$
- $\bot^\mathcal{I} = \emptyset$
- $\neg C^\mathcal{I} = \Delta^\mathcal{I} - C^\mathcal{I}$
- $(C \cap D)^\mathcal{I} = C^\mathcal{I} \cap D^\mathcal{I}$
- $(C \cup D)^\mathcal{I} = C^\mathcal{I} \cup D^\mathcal{I}$
- $(\forall R.C)^\mathcal{I} = \{a \in \Delta^\mathcal{I} | \forall b. (a.b) \in R^\mathcal{I} \rightarrow b \in C^\mathcal{I}\}$
- $(\exists R.C)^\mathcal{I} = \{a \in \Delta^\mathcal{I} | \exists b. (a.b) \in R^\mathcal{I} \land b \in C^\mathcal{I}\}$

Another way of giving semantics to $\mathcal{ALC}$ is by giving a correspondence with first order logic (FOL). We can translate every concept definition into an equivalent FOL. Atomic concepts and roles are mapped onto $A(\alpha)$ and $R(\alpha, \beta)$ respectively. A concept $C$ and a role $R$ correspond to the FOL formulae $F_C(\alpha)$ and $F_R(\alpha, \beta)$ where $\alpha$ and $\beta$ denote free variables..
**Definition 14** (Transformational Semantics of $\mathcal{ALC}$). We define the interpretation $\mathcal{I}$ as follows:

\[
\begin{align*}
\top^\mathcal{I} &= \text{true} \\
\bot^\mathcal{I} &= \text{false} \\
\neg C^\mathcal{I} &= \neg F_C(\alpha) \\
(C \sqcap D)^\mathcal{I} &= F_C(\alpha) \land F_D(\alpha) \\
(C \sqcup D)^\mathcal{I} &= F_C(\alpha) \lor F_D(\alpha) \\
(\forall R.C)^\mathcal{I} &= \exists x. F_R(\alpha, x) \land F_C(x) \\
(\exists R.C)^\mathcal{I} &= \forall x. F_R(\alpha, x) \rightarrow F_C(x)
\end{align*}
\]

Since we can give this transformational grammar it becomes clear that $\mathcal{ALC}$ is a subset of FOL.

**A.2 Terminologies**

The previous section showed the basic ingredients that allow us define a concept. In order to describe a domain of interest we need to specify how a group of concepts relate to each other. The description of how a set of concepts relate to each other is called a terminology or T-Box. A terminology is defined as follows:

**Definition 15** (T-Box). We define a T-Box $\text{TBox}$ as follows:

- **Inclusion**: $C \sqsubseteq D$ with the interpretation $C^\mathcal{I} \subset D^\mathcal{I}$, meaning that the concept $C$ is a subclass of $D$.
- **Equality**: $C \equiv D$ with the interpretation $C^\mathcal{I} = D^\mathcal{I}$, meaning that $C$ can be defined as $D$.
- **Definition**: An equality whose left hand side is atomic.
- **Unique**: No symbol is defined more than once.
- **A-Cyclic**: No definition is (indirectly) defined in terms of itself.

Concepts that only occur on the right hand side of definitions are called primitive concepts. Notice that inclusion $C \sqsubseteq D$ can be expressed by introducing a new base concept $\bar{C}$ as $C \equiv \bar{C} \cap D$. The idea is that $\bar{C}$ exactly captures the
difference between a \( \bar{C} \) and a \( D \). The inclusion does not really make a language more expressive, it merely is a convenient way of describing how one concept is more generic than another.

Notice that a T-Box as defined above can always be expanded into a T-Box where every definition is defined in terms of only base symbols. The expansion procedure recursively substitutes every defined concept occurring in a definition with its defining expression. This procedure can generate a T-Box that is exponential in size. In \[69\] it is shown that this blowup is polynomial if one imposes some restrictions.

A.3 Extensional Knowledge

Extensional knowledge deals with actual individuals that are instances a specific concept. For example we could state that the individual \textit{mathilda} is a person. Traditionally this is written as Person(Mathilda). An A-Box (ABox) is a collection of such assertions.

In order to deal with individuals we extend the interpretation function \( I \) in such a way that for every individual \( a \) we have that \( a^I \in \Delta^I \). It is assumed that if \( a \) and \( b \) are distinct names then \( a^I \neq b^I \).

Concept membership \( C(a) \) can now be interpreted as \( a^I \in C^I \) and role fulfillment as \( (a^I, b^I) \in R^I \).

A.4 Reasoning

There are basically two forms of reasoning that are of relevance for description logic. Reasoning with intensional knowledge and reasoning with extensional knowledge.

Reasoning with intensional knowledge deals with inferring relations that are not defined but can be extracted from the description. The basic inference on concept expressions is subsumption, written as \( C \sqsubseteq D \). We check if the first concept always is a subset of the second concept.
Subsumption is used in order to classify a knowledge base. The aim is to infer which concepts subsume one another. Consider figure 2–1 in the previous chapter. We can derive that a mother is a woman, since both mother and woman are subclass of female and person.

Subsumption forms the base for extracting all other forms of relevant information. The notion of satisfiability (i.e., assuring that every concept allows at least one individual). A concept makes a knowledge base inconsistent if it subsumes to the bottom concept which allows no individuals (i.e., the concept $C$ is inconsistent whenever: $C \sqsubseteq \bot$).

The third form of reasoning is used to establish whether two concepts are mutually exclusive. Meaning that it is not possible to find an instance of $C$ that is an instance of $D$ as well. Disjointness can be computed by observing that $C \cap D \sqsubseteq \bot$.

For extensional knowledge we have two forms of inference that are of relevance. First we have the problem of instance classification. That is given an individual $a$ we verify if $a$ is an instance of that specific concept. Second is the retrieval problem. Given a concept expression $C$ which individuals $a \in \text{ABox}$ in the knowledge base are an instance of this concept.

To sum things up, we have the following tools for reasoning available:

- Subsumption: $C \sqsubseteq D \Leftrightarrow C^I \subseteq D^I$
- Equality: $C = D \Leftrightarrow C^I = D^I$
- Disjointness: $C$ and $D$ are disjoint when $C^I \cup D^I = \varnothing$
- Inconsistent: $C = \bot$
- Membership: $C(a) \Leftrightarrow a^I \in C^I$
- Retrieval: The retrieval of $C$ is the set $\{a \mid a \in \text{ABox} \land C(a)\}$
As shown above all of these can also be expressed using subsumption, so any reasoning engine supporting subsumption can be used to verify the other relationships as well.

### A.5 Open-World versus Closed-World Assumption

One could say that a knowledge base is very similar to a database. The intensional knowledge represents the database schema and the extensional knowledge represents the data items contained within a database. This comparison is natural, but there is however a key difference between a database and a description logic. This difference lies in the approach both systems take to absence of facts. When data is absent it might not be possible to deduce whether a certain fact is true or not. In general this has lead to two approaches to dealing with the absence of information:

- **Closed World**: If it is impossible to prove whether a fact $P$ or $\neg P$ is true, we assume that $\neg P$ is true.

- **Open World**: If it is impossible to prove whether a fact $P$ or $\neg P$ is true we consider the fact to be undetermined for the time being.

Suppose we have the following statement:

*Erwin is a citizen of the Netherlands*

And the following question:

*Is Erwin a citizen of the USA*

Under the closed world assumption the only conclusion we can draw is No, since the absence of the information implies that Erwin is not a citizen of the USA. If we use the open world assumption however this cannot be concluded, and the answer would be unknown, since Erwin could have a dual citizenship.

Another classic example is:

*A Person can have only one Mother,*

*The mother of Erwin is Maria*
Suppose we add the statement: *The mother of Erwin is Mieke* to our knowledge base. Under the closed world assumption this would result in an exception, since a Person (*Erwin*) can only have one mother. Under the open world assumption the system would derive that *Maria* and *Mieke* are the same person. Which happens to be the case in this particular example.

Databases normally operate under the closed world assumption, whereas description logics operate using the open world assumption. The two different approaches have both their advantages and drawbacks. The open world assumption is more expressive, since if the fact \( P \) is absent we must consider a model where \( P \) is true and a model where \( \neg P \) is true. Consider the example given in [59].

Suppose we have the following knowledge base:

\[
\begin{align*}
\text{hasChild}(JOCASTA, OEDIPUS) \\
\text{hasChild}(OEDIPUS, POLYNEIKES) \\
\text{hasChild}(JOCASTA, POLYNEIKES) \\
\text{hasChild}(POLYNEIKES, THERSANDROS) \\
\text{Patricide}(OEDIPUS) \\
\neg \text{Patricide}(THESANDROS)
\end{align*}
\]

The knowledge base is shown in Figure A–1. Suppose we want to know whether or not \((\exists \text{hasChild.} (\text{Patricide} \land \exists \text{hasChild.} \neg \text{Patricide}))(JOCASTA)\) holds, i.e the answer to the question whether Jocasta has at least two children, one which killed his or her father and one that did not.

We cannot conclude this to be the case for Jocasta if we use the closed world assumption. This follows from the fact that it is unknown whether or not Polyneikes killed his father.

Under the open world assumption however there are two possible models, (as shown in Figure A–1). One world in which Polyneikes is a patricide and one in
which he is not. For both cases we can establish a model that makes the sentence true. Hence we conclude this to be the case for Jocasta.
REFERENCES


BIOGRAPHICAL SKETCH

Erwin was born in Maarheeze, a small village in the province Brabant in the Netherlands. He attended the University of Utrecht and obtained a master of arts degree in computer science. Erwin’s main research interests are pervasive computing, artificial intelligence, protocol verification and searching in peer-to-peer systems.