A TASK EXECUTION FRAMEWORK FOR AUTONOMIC UBIQUITOUS COMPUTING

BY

ANAND RANGANATHAN

B.Tech., Indian Institute of Technology, Madras, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

A TASK EXECUTION FRAMEWORK FOR AUTONOMIC UBIQUITOUS COMPUTING

Anand Ranganathan, PhD.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2005
Prof. Roy H. Campbell, Advisor

Ubiquitous Computing advocates the enhancement of physical spaces with computing and communication resources that help users perform various kinds of tasks. However, these environments are highly complex, dynamic and fault-prone. Besides, different environments have different resources, architectures and policies. Hence, developers find it difficult to program new applications and services in these rich environments. In this thesis, we propose a framework for autonomic ubiquitous computing, that enables these environments to be self-configuring, self-repairing, and adaptive. The framework allows developers to program ubiquitous computing environments in terms of high-level, parameterized tasks. The values of task parameters may be provided by the end-user or automatically inferred by the framework based on the current state of the environment, context-sensitive policies, and learned user preferences. The framework uses a novel semantic discovery process to infer the best values of parameters. It can also recover from failures of one or more actions by using alternative resources. In order to evaluate this approach, we define metrics for measuring the complexity of developing and managing systems. We then evaluate the framework based on these metrics and also describe our experiences while using the system.

Acknowledgements

I'd like to thank my advisor, Prof. Roy Campbell for his unstinting support, his creative ideas, his sense of humor and for the many wacky discussions we had. I'd also like to thank Profs. M. Dennis Mickunas, Dan Roth, Steve LaValle and Klara Nahrstedt for their guidance and support. I'm grateful to Anda for helping me get a few precious moments of my advisor's time and helping me navigate the various administrative mazes. I'd like to thank my numerous colleagues at SRG and Co, both past and present - Manuel, Chris, Jalal, Chetan, Geta, Robert, Apu, Prasad, Zahid, Fabio, Renato, Christina, Suvda, Cigdem, Dulcineia, Bhaskar, Pradeep, Gaurav, Prashant, Binny et al - for their help and for providing company during the lonely plough through the PhD fields. I'd also like to thank Perry, Jon, Maria and Dave for abstracting the low-level system administration details from much of my work. I'm also grateful to the NSF for financial support.[1]

I'd like to thank my parents for their support, patience and guidance while I churned away in the great plains of the U.S. I'd like to thank my friends from various social circles who made Champaign sparkling and my time here memorable. Finally, I'd like to thank Champaign-Urbana for being a fine home for 5 eventful years, while at the same time motivating me to finish my PhD as fast as possible.

[1]NSF CCR 0086094 ITR

Contents

List of Tables . . . . **viii**

List of Figures . . . . **ix**

Chapter

1 Introduction **1**

2 Background - Active Spaces and Gaia **5**

3 Task Model **8**

3.1 Example of a Task 8

3.2 Task Parameters 12

3.3 Ontologies of Task Parameters 13

3.4 Activities in a Task 17

3.5 Composing Activities to form Tasks 18

3.6 Developing a Task 20

3.7 Other Examples of Tasks 20

3.7.1 A Lecture Task 20

3.7.2 A Music Playing Task 21

3.7.3 A Collaboration Task 23

3.7.4 A Notification Task 23

4 Developing Activities using the Olympus Programming Model 25

4.1 High-Level Operands 25

4.2 High-Level Operators 30

4.3 Expressiveness of the Olympus Programming Model 31

4.3.1 Characterizing the lifecycle of entities 31

4.3.2 Characterizing the state-space of entities 32

4.3.3 Characterizing the relationship-space between entities 32

4.3.4 Operators for a ubiquitous computing environment 34

4.4 Comparison with traditional programming models 36

5 Why the Task Model? - Exploring the Space of Automation 37

6 The Task Execution Framework 41

6.1 Architecture 41

6.2 Executing a Task 42

6.3 Discovering Possible Parameter Values 43

6.3.1 Semantic Matching of Classes 46

6.3.2 Checking Prolog Policies 48

6.3.3 Optimizing Task Execution 49

6.4 Self-Repair 52

6.5 The Learning Framework 54

7 Implementation Details and Evaluation 57

7.1 Evaluating the expressiveness of the Framework 58

7.1.1 Evaluating the expressiveness of the Olympus Programming Model 58

7.1.2 Evaluating the expressiveness of the Task Model 59

7.2 Performance 66

7.3 Time Complexity 68

7.4 Evaluating task performance by an end-user 69
7.5 Other Experiences with the System 70

8 Development Complexity Analysis 75
8.1 Measuring Development Complexity 75
8.1.1 Task-Structure Complexity 76
8.1.2 Unpredictability 79
8.1.3 Size Complexity 80
8.1.4 Chaotic Complexity 80
8.1.5 Algorithmic Complexity 82
8.1.6 Why these metrics? 84
8.2 Features of the Task Execution Framework that help reduce Complexity 86
8.2.1 Self-Configuration and Self-Repair 87
8.2.2 High-Level Programming 87
8.2.3 Ontologies for Organization of Concepts and Defining Semantics 88
8.3 Evaluating the Reduction in Complexity 89
8.4 The Complexity-Flexibility Trade-off 92
8.5 Distinguishing between the complexity of a system and the complexity of using a system 93

9 Related Work 95

10 Future Work 98

11 Conclusions 101

Appendix

A Example slideshow task in C++ consisting of various activities . . . 103

B Segment of Parameter XML File for a slideshow task 106

C Defining the Learning Feature Space **109**

D Defining Labels of Learning Feature Values **114**

E Learning Training File **117**

References **119**

Author's Biography **124**

List of Tables

4.1 Segment of Parameter XML File for migrating a slideshow 29

4.2 Lua script with parameter-obtaining and world-altering activities . . 30

7.1 Basic Operands supported by Olympus 60

7.2 Some lifecycle operators supported by Olympus 61

7.3 Some state-space operators supported by Olympus 62

7.4 Some relationship-space operators supported by Olympus that operate on the relationship between a target entity operand and the argument operand 63

7.5 Some more relationship-space operators supported by Olympus 64

7.6 Some event operators supported by Olympus 65

7.7 Comparison of times taken for tasks in different configurations 67

7.8 Time taken by user to perform various slideshow and music tasks . . 71

7.9 Time taken by user to perform various notification tasks 72

8.1 Comparison of Task-Structure Complexity (TSC) complexity and Cognitive Algorithmic Complexity (CAC) with and without support of our framework 91

List of Figures

2.1 Application Framework and slide-show application components (in parentheses) . . . 7

3.1 Slideshow task represented as a flowchart . . . 10
3.2 Screenshot of Task Control GUI . . . 11
3.3 Presentation Hierarchy . . . 14
3.4 Controller Hierarchy . . . 15
3.5 Device Hierarchy . . . 15
3.6 Task Structure . . . 18
3.7 Lecture task represented as a flowchart . . . 22

4.1 Different relations between regions, as defined by RCC . . . 33
4.2 Relationships between different kinds of Active Space Entities expressible in Olympus . . . 35

6.1 Architecture of Task Execution Framework . . . 42
6.2 Discovery Process . . . 45

8.1 Different Aspects of System Complexity . . . 77
8.2 Task graphs with different task-structure or cyclomatic complexities . . . 78
8.3 The Complexity-Flexibility Tradeoff . . . 93

Chapter 1

Introduction

Ubiquitous Computing envisions a world with users interacting naturally with device-rich, context-aware environments to perform various kinds of tasks. Over the past few years, different ubiquitous computing scenarios and applications have been proposed and various prototype systems supporting these scenarios have been built. As we move forward to more widespread deployment of ubiquitous computing, we face a number of challenges arising from the complexity of ubiquitous computing environments. These environments feature large mobile and distributed systems containing a variety of inter-dependent services and devices. They are also highly dynamic and fault-prone. Thus, programs running in these environments must be able to adapt to the changing contexts and resource availabilities, and recover from failures of devices, services and applications. Different environments may have different resources and hence, many programs are not portable across environments. Developers, often, have to re-develop their applications and services for new environments. Finally, different environments may have different policies regarding the usage of resources for performing various kinds of tasks. Developers have to be aware of these policies while developing their services and applications. These issues place a bottleneck on the rapid development and prototyping of new services and applications in these environments.

A new challenge introduced by ubiquitous computing is that since these environ-

ments are characterized by a large number of different types of devices, services and applications, there are often different ways of performing the same task, using different resources or different strategies. However, some ways are better than others depending on the current context, resources available and user preferences. Hence, programs need to choose the "best" way of performing a task from the various choices available. The developer, though, should not be burdened with this task.

In our previous research, we have developed prototype ubiquitous computing environments that are programmed and managed with the help of a middleware called Gaia[13]. However, when we tried to deploy these environments in a wide-scale across different rooms and buildings, we encountered several challenges arising from the complexity, dynamism and heterogeneity of the environments. The promise of ubiquitous computing environments will not be realized unless these systems can effectively "disappear". In order to do that, they need to become autonomous by managing their own evolution and configuration without explicit user or administrator action. In this thesis, we propose a framework that enables autonomic ubiquitous computing, i.e. it allows ubiquitous computing environments to be self-configuring and adaptive, and to require minimal user intervention. The framework allows developers and administrators to program and configure ubiquitous computing environments in terms of high-level tasks. A task is a set of actions performed collaboratively by humans and the ubiquitous system to achieve a goal. Common tasks in a smart conference room, for instance, include displaying slideshows, giving lectures, playing music, collaborating with others, and migrating applications.

In this framework, developers first develop primitive activities that perform actions like starting, moving or stopping components, changing the state of devices, services or applications or interacting with the end-user in various ways. They then develop programs or workflows that compose a number of primitive activities into a task that achieves a certain goal.

Each task is associated with one or more parameters that influence how it is performed. The parameters may be devices, services or applications to employ while performing the task or they may be strategies or algorithms to use. For example, even the relatively simple task of displaying a slideshow has a number of parameters like the devices and applications to use for displaying and navigating the slides, the name of the file, etc. When the task is executed, the task execution framework obtains the values of the different parameters in the task by either asking the end-user or by automatically deducing the best value of the parameter. The best value is deduced based on constraints specified by the developer, the current state of the environment, context-sensitive policies, and user preferences. Hence, a task can be executed entirely automatically (where the environment decides all the parameter values), entirely manually (where the end-user chooses all parameters), or anywhere in the space in-between. The framework, thus, frees developers and end-users from the burden of choosing myriad parameter values for performing a task, although it does allow end-users to override system choices and manually configure how the task is performed.

The framework can also recover from failures of one or more actions by using alternate resources. While executing the actions, it monitors the effects of the actions through various feedback mechanisms. In case any of the actions fail, it handles the failure by re-trying the action with a different resource.

The main contributions of the thesis are the proposal of a high-level task specification model and an autonomic task execution framework for ubiquitous computing environments. The framework enables self-configuration, self-repair and rapid prototyping of new tasks. It also enhances portability since tasks are written in a high-level manner, in terms of abstract resources or parameters and are not strongly tied into the characteristics of any particular environment. Some of the key elements of the framework are:

1. Ontologies for specifying hierarchies of different kinds of entities and their properties in a standard, reusable way.

2. A novel semantic discovery algorithm based on ontological hierarchies for discovering appropriate resources for performing a certain task.

3. Context-sensitive policies, written as Prolog rules, that are checked while choosing appropriate resources.

4. A multi-dimensional utility function for choosing the "best" values of different task parameters.

We have implemented a prototype task development and execution framework and have incorporated it in our middleware for ubiquitous computing. In the following chapters we describe our planning model and framework in detail. Chapter 2 describes the underlying Gaia infrastructure and our vision and implementation of Active Spaces. Chapter 3 describes the task programming model. Chapter 4 describes our high-level programming model, called Olympus, using which tasks are developed. Chapter 5 explores the space of automation and explains why we designed our task model the way we designed it. Chapter 6 has details on the architecture and the process of executing, optimizing and repairing tasks. We evaluate the framework in Chapter 7. Chapter 8 analyzes the framework based on the reduction in development complexity it enables. Chapters 9, 10 and 11 have related work, future work and our conclusions.

Chapter 2

Background - Active Spaces and Gaia

The Task Execution Framework has been implemented for Active Spaces on top of the Gaia middleware. An Active Space is a physical space, such as a room or a building, that has been enhanced with a large number of digital devices such as various sensors, computers and actuators. The Gaia middleware [13], manages the resources of an Active Space, and allows programming the Active Space. Gaia contains a set of core services that manages the resource collection and provides a programming interface to application developers. It supports an application framework that decomposes an Active Space application into smaller components that can be migrated across various devices in an active space and adapted to the requirements of a space. Gaia uses CORBA [37] to enable distributed computing.

The core services of Gaia include a Presence Service and a Space Repository that together help in discovering entities (i.e. applications, devices and services) present in the space. The Presence Service detects the presence of entities using heartbeats that are sent by all Gaia entities. The Space Repository maintains a list of all active entities present in a space along with their descriptions. The Space Repository has a MySQL database backend and allows queries on the entities present in the space.

Gaia also has a context infrastructure[36] that allows sensing and inferring different types of contexts. There are many different kinds of context information - these include physical contexts (like location of people, things or software objects, time), environmental contexts (weather, light and sound levels), user contexts (activity, mood, schedule), system contexts (network traffic, status of printers) and so on. All these contexts may influence the way a user performs a certain kind of task in an Active Space. Gaia also uses a role-based access control mechanism. Access control policies are used to decide if a certain principal (i.e. an object or a user) has access to a certain resource in a certain context.

Gaia uses ontologies in DAML-OIL and OWL to define the semantics of different concepts like context information, applications, services, devices, file-types, roles, etc. [38]. Ontologies [26] have been widely used in many areas such as knowledge and content management, electronic commerce and the Semantic Web [25]. They allow defining hierarchies and properties of entities in a standard, reusable and portable manner. Gaia also has a location middleware [32] that allows reasoning about location data obtained from different kinds of sensors and capturing the uncertainty in location information.

Applications in Gaia are built using an extension of the Model-View-Controller framework [35],[5]. Applications are made up of five components: model, presentation (generalization of view), adapter, controller and coordinator. The framework is shown in Figure 2.1. The ApplicationComponent entity type refers to these five components, while the Application entity type is a composite type referring to the set of the components belonging to a single application.

The model implements the logic of the application and exports an interface to access and manage the application's state. Controllers act as input interfaces to the application and presentations as output interfaces. The adapter maps controller inputs into method calls on the model. The coordinator manages the composition of

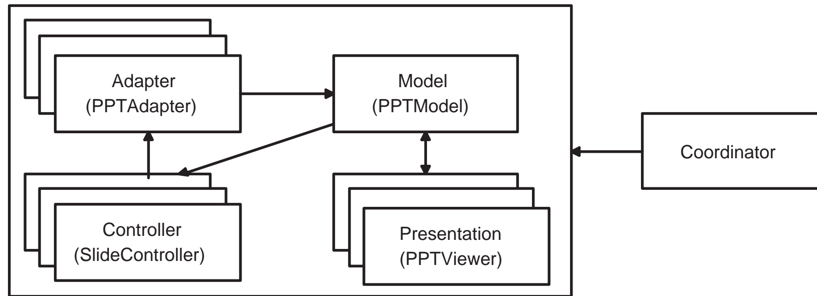

Figure 2.1: Application Framework and slide-show application components (in parentheses)

the other components of the framework. For example, in a slideshow application, the model (a PPTModel component) maintains the name of the file and the current slide number; the presentation (a PPTViewer component) actually renders the slides; and the controller (a SlideController component) allows navigating through the slides.

Chapter 3

Task Model

In this section, we describe our model of tasks, starting with an example and then looking at the structure and parameters of tasks, as well as ontology support

3.1 Example of a Task

In order to motivate the need for autonomic ubiquitous computing, we give an example of the task of displaying a slideshow. The wide variety of devices and software components available in a ubiquitous computing environment allows different ways of configuring the slideshow application. Our prototype smart room, for instance, allows presentations to be displayed on large plasma displays, a video wall, touch screens, handhelds, tablet PCs, etc. The presentation can be controlled using voice commands (by saying "start", "next", etc.) or using a GUI (with buttons for start, next, etc.) on a handheld or on a touch-screen. Different applications (like Microsoft PowerPoint or Acrobat Reader) can be used as well for displaying the slides.

Hence, in order to display a slideshow, appropriate choices have to be made for the different devices and components needed in the task. Developers of slideshow tasks may not be aware of the devices and components present in a certain environment and hence cannot decide before-hand what is the best way of configuring the task.

End-users may also not be aware of the different choices and they may also not know how to configure the task using different devices and components. Besides, access to some devices and services may be prohibited by security policies. Finally, components may fail due to a number of reasons.

In order to overcome these problems, our task execution framework allows developers to specify how the slideshow task should proceed in a high-level manner. Developers specify the different activities involved in the task and the parameters that influence how exactly the task is executed. These parameters include the devices and components to be used in the task, the name of the file, etc. They can also specify constraints on the value of the parameters. For instance, they can specify that only plasma screens are to be used for displaying slides. For each parameter, the developer can also specify whether the best value is to be deduced automatically or obtained from the end-user. Fig 3.1 shows a portion of the overall control flow of the slideshow task represented as a flowchart. The task execution framework takes care of executing the different activities in the task, discovering possible values of the parameters and picking the best value on its own or asking the end-user for the best value.

The result of this slideshow task is an application that displays the required file. This application is based on a distributed Model-View-Controller framework [5] and consists of the following distributed components: a model (which manages application state), one or more presentations (that provides a view of the state), one or more controllers (that allow users to control the application) and a coordinator (that has information about the structure of the application and its components).

The framework also simplifies the performance of tasks for end-users. End-users interact with the framework through a Task Control GUI. This GUI runs on a fixed display in our prototype smart room or may also be run on the user's laptop or tablet PC. The GUI displays a list of tasks that have been developed for the smart

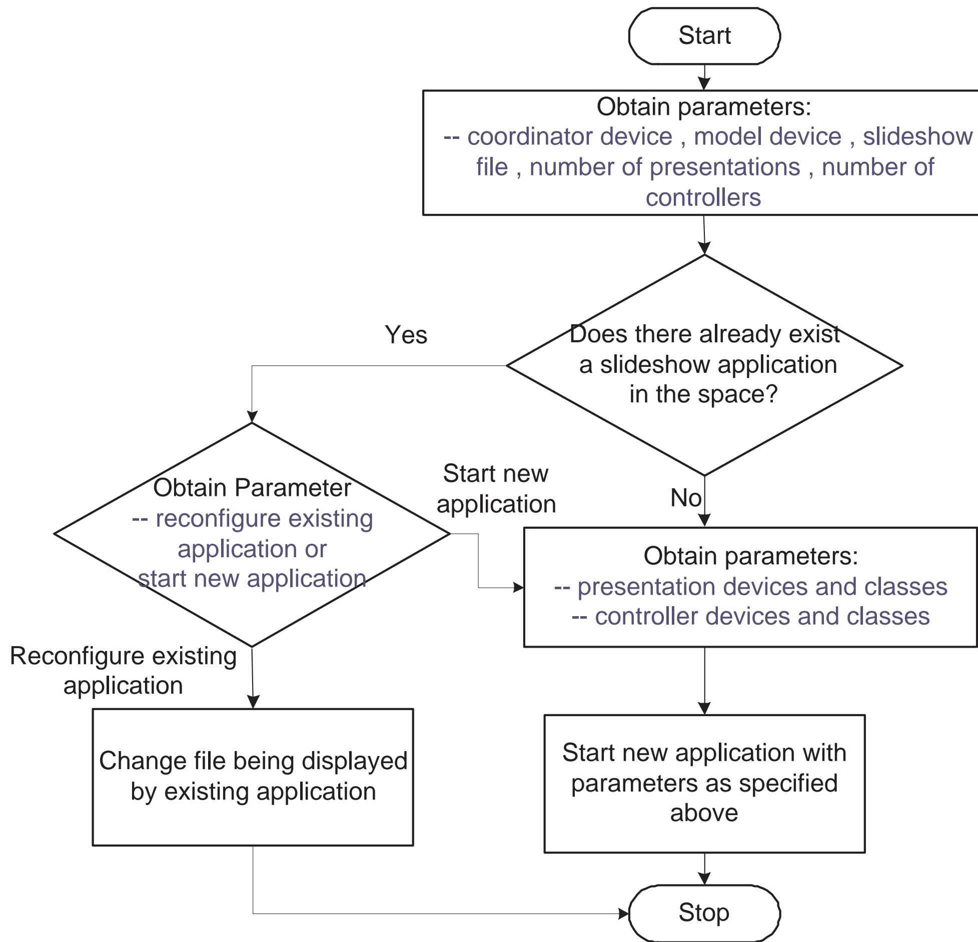

Figure 3.1: Slideshow task represented as a flowchart

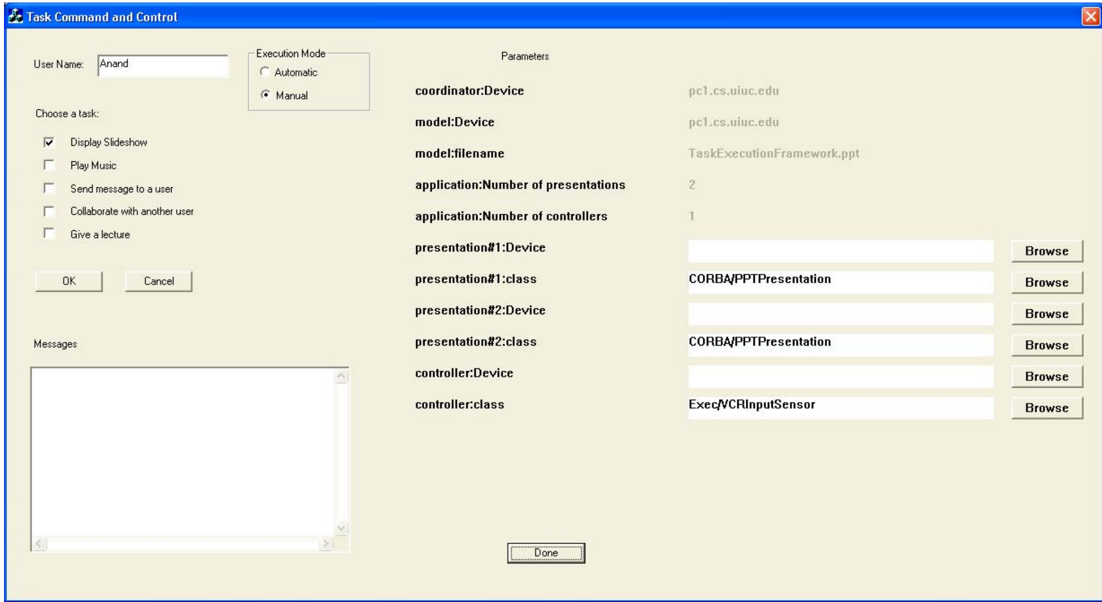

Figure 3.2: Screenshot of Task Control GUI

room. The end-user enters his name and indicates the task he wants to perform (like "Display Slideshow", "Play Music", etc.). The framework, then, presents him with a list of various parameters. In the case of parameters that have to be obtained from the end-user, the user enters the value of the parameter in the edit box next to the parameter name. In the case of automatic parameters, the framework discovers the best value and fills the edit box with this value. The user can change this value if he desires. For both manual and automatic parameters, the user can click the "Browse" button to see possible values of the parameter and choose one.

Fig 3.2 shows the Task Control GUI in the middle of the slideshow task. The presentation and controller parameters need to be obtained in the current activity. The values of the first five parameters (coordinator, model and application parameters) have already been specified by the user or obtained automatically in a previous activity. The Task Execution Framework has automatically found the best values of the presentation and controller classes and it presents them to the user. The presentation and controller device parameters have to be provided by the end-user. The GUI also provides feedback to the user regarding task execution and if any failures occur.

3.2 Task Parameters

The parameterization of tasks helps make them flexible and adaptive. The explicit representation of different parameters of the task allows the task execution framework to obtain the values of the parameters using different mechanisms and customize the execution of the task depending on the current context and the end-user. There are two kinds of task parameters: behavioral parameters, which describe which algorithm or strategy is to be used for performing the task; and resource parameters, which describe which resources are to be used.

Each task parameter is associated with a class defined in an ontology. The value of the parameter must be an instance of this class (or of one of its subclasses). For example, the filename parameter for a slideshow task must be the "SlideShowFile" class (whose subclasses are files of type ppt, pdf or ps). Each task parameter may also be associated with one or more properties that constrain the values that it can take.

The different parameters for the various entities in a task are specified in an XML file. Appendix B shows a segment of the parameter XML file for the task of displaying a slideshow. The XML file specifies the name of the parameter, the class that its value must belong to, the mode of obtaining the value of the parameter and any properties that the parameter value must satisfy. In case the parameter value is to be inferred, automatically, by the framework, the XML file also specifies the metric to use for ranking different possible parameter values. For example, the XML file in Appendix B defines two parameters for the model of the slideshow application - the device on which the model is to be instantiated and the name of the file to display. The device parameter should be of class "Device" and is to be automatically chosen by the framework using the space policy. The filename parameter should be of class "SlideShowFile" and is to be obtained from the end-user. Similarly, some other parameters of the slideshow task are the number of presentations, the number

of controllers, devices and classes of the different presentation components.

Task parameters are similar to parameters passed to methods in object-oriented languages like smalltalk and C++. They are of a certain data-type or class and the value that is passed may belong to that class or any of its subclasses (thanks to polymorphism). One difference between a task parameter and any other object-oriented method parameter is that the value of the task parameter is resolved explicitly in the task program by calling a "queryForTaskParameters" function call. The value of an object-oriented method parameter is, however, resolved at the beginning of the method or upon first use, depending on the exact parameter passing mechanism used. Task parameters are also defined externally in the parameter XML file. This allows developers and administrators to change the class or properties of the parameters easily. Also, the value of a task parameter may be obtained by either asking the end-user or by using a powerful semantic discovery process. This semantic discovery process involves an intelligent type coercion, where the best class of the task parameter value is first discovered. After that, the best instance of this best best class is then discovered.

3.3 Ontologies of Task Parameters

In order to aid the development of tasks and to have common definitions of various concepts related to tasks, we have developed ontologies in OWL [20] that describe different classes of task parameters and their properties. There are eight basic classes of task parameters: Application, ApplicationComponent, Device, Service, Person, PhysicalObject, Location and ActiveSpace. These basic classes, further, have subclasses that specialize them. We briefly describe the ApplicationComponent hierarchy in order to illustrate the different kinds of hierarchies.

Fig 3.3 shows a portion of the hierarchy under ApplicationComponent describ-

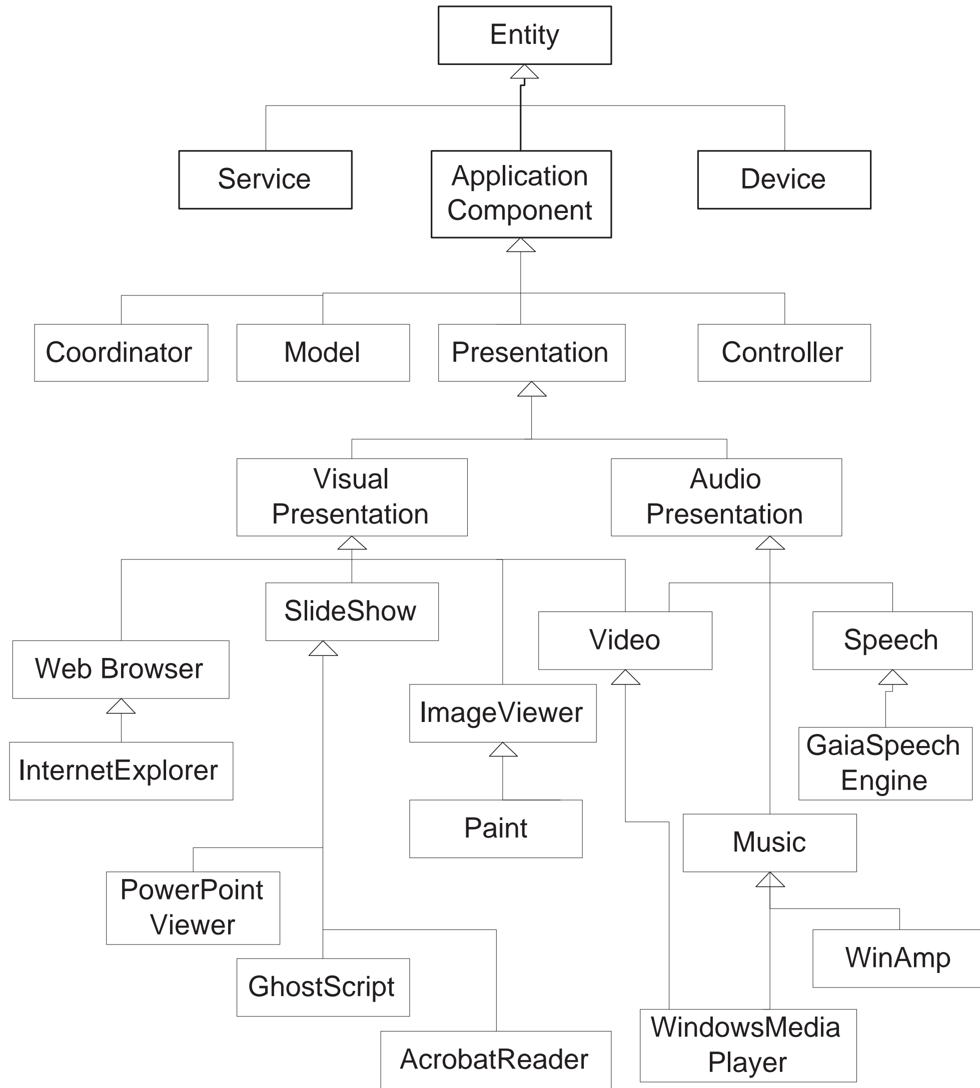

Figure 3.3: Presentation Hierarchy

ing different kinds of Presentation components. The hierarchy, for instance, specifies two sub-classes of "Presentation" - "Visual Presentation" and "Audio Presentation". It also further classifies "Visual Presentation" as "Web Browser", "ImageViewer", "SlideShow" and "Video". Ontologies allow a class to have multiple parents - so "Video" is a sub-class of both "Visual Presentation" and "Audio Presentation". Similarly, Fig 3.4 shows different kinds of Controller components and Fig 3.5 shows a portion of the device hierarchy.

The ontologies also define properties of these classes. An example of a property

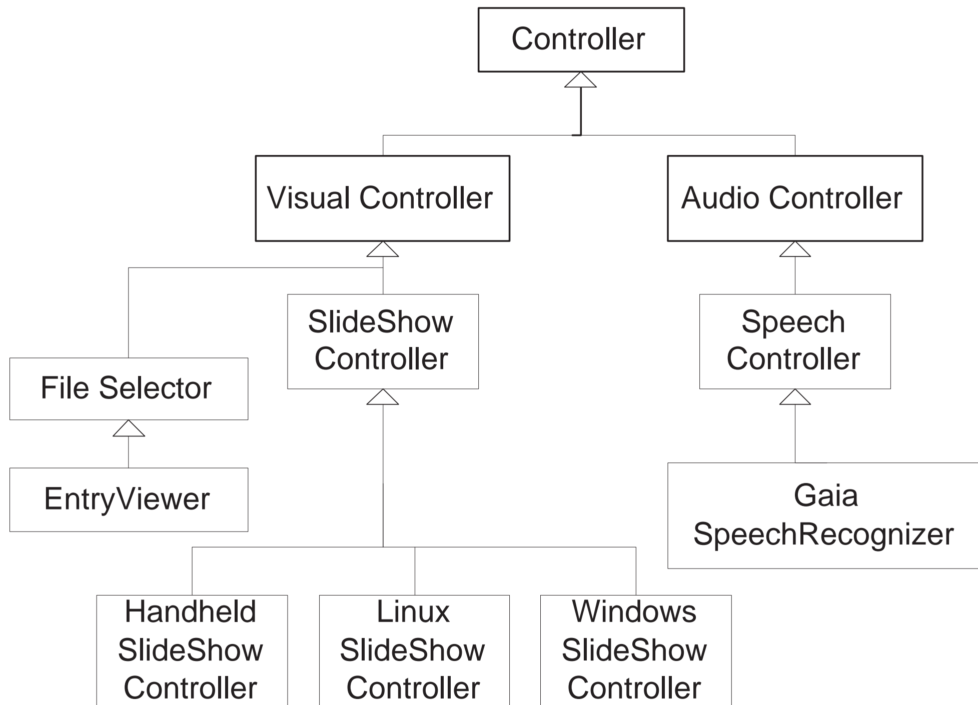

Figure 3.4: Controller Hierarchy

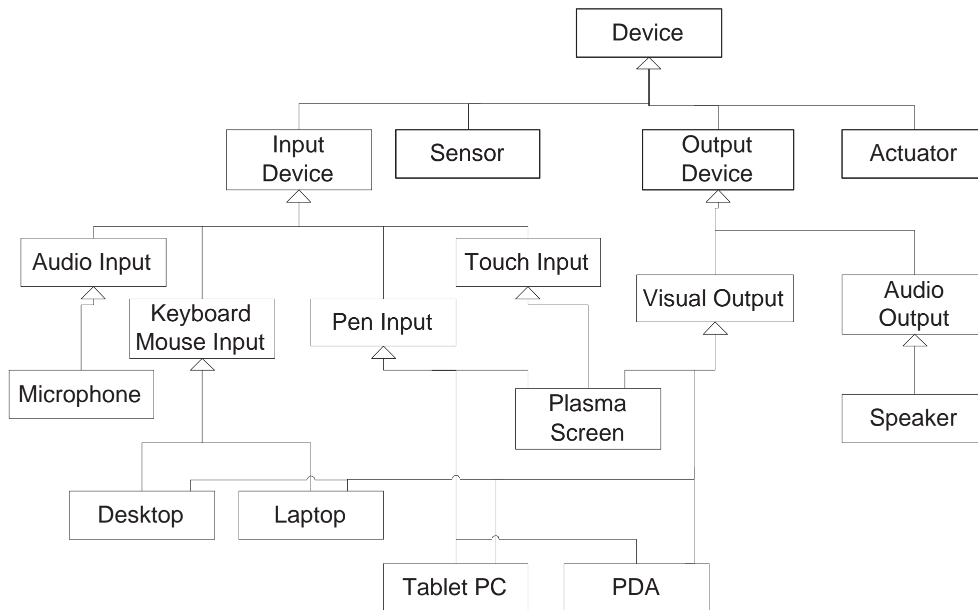

Figure 3.5: Device Hierarchy

is the *requiresDevice* relationship which maps application components to a Boolean expression on devices. For example,

$$requiresDevice(PowerPointViewer) = \\ PlasmaScreen \vee Desktop \vee Laptop \vee TabletPC$$

This means that the *PowerPointViewer* can only run on a *PlasmaScreen*, *TabletPC* or a *Desktop*. Another relation, *requiresOS*, maps application components to operating systems. E.g.

$requiresOS(PowerPointViewer) = Windows$

The ontologies are initially created by an administrator. As new applications, devices and other entities are added to the environment, the ontologies are extended by the administrator or application developer to include descriptions of the new entities.

Ontologies are very useful for specifying the types and properties of task parameters. They are more expressive than several other typing systems like object oriented class hierarchies. Simple class hierarchies cannot express new kinds of relationships between different classes (apart from the "subclass-of" relationship). Ontologies, however, can define other kinds of relationships between classes. They also have a foundation in description logic. Hence, it is possible to perform different kinds of reasoning on ontologies such as classification, satisfaction and subsumption based reasoning. It also possible to check the consistency of ontologies, and make sure that different assertions in the ontologies do not conflict. Finally, ontologies can be reused and exchanged between different parties easily, since they are written in a standard Semantic Web language like RDF or OWL, which are both based on XML. Also, theoretically, it is possible to interoperate different systems by finding mappings between the various concepts defined in their ontologies.

Ontologies play a key role in our task execution framework in enabling high-level programming. Developers program the environment using concepts that are high

up in the ontology hierarchy. The task execution framework then maps the high-level concepts into low-level concepts depending on the task being performed, the current state and context of the environment and the users' preferences. The low-level concepts are typically leaf concepts in our framework.

3.4 Activities in a Task

Tasks are made up of a number of reusable sub-tasks or activities, that can be recombined in different manners. Different tasks often have common or similar activities; hence it is easy to develop new tasks by reusing activities that have already been programmed.

There are three kinds of activities allowed in our framework: parameter-obtaining, state-gathering and world-altering (Fig 3.6). Parameter-obtaining activities involve getting values of parameters by either asking the end-user or by automatically deducing the best value. State-gathering activities involve querying other services or databases for the current state of the environment. World-altering activities change the state of the environment by creating, re-configuring, moving or destroying other entities like applications and services.

In parameter-obtaining activities, developers list various parameters that must be obtained. The descriptions of these parameters are in the task parameter XML file (such as the one in Appendix B). In the case of parameters obtained from the end-user, the task execution framework contacts the Task Control GUI. In case of parameters whose values must be deduced automatically, the framework contacts a Discovery Service to get the best value. Further details of the discovery process are in Chapter 6.

World-altering and state-gathering activities are written in the form of C++ functions. These activities can have parameters. They are developed using a high-level

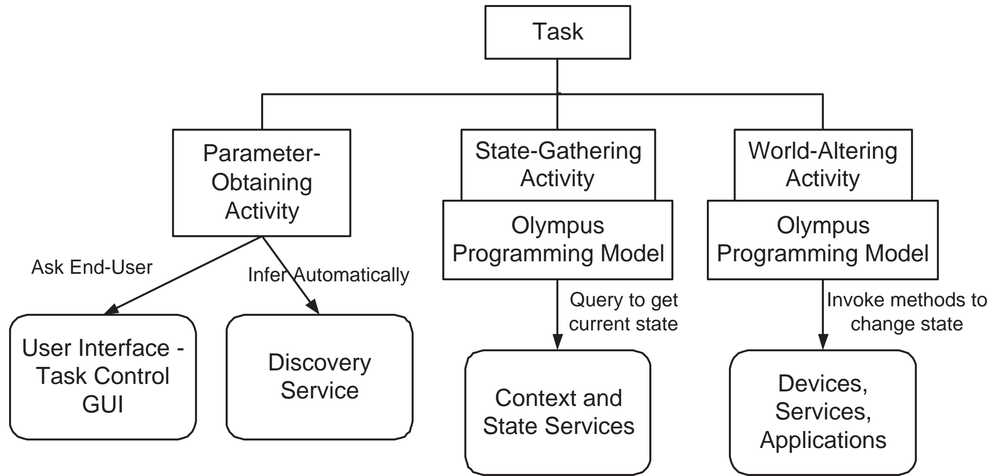

Figure 3.6: Task Structure

programming model called Olympus [16]. The main feature of this model is that it represents common ubiquitous computing operations as high-level operators. Examples of operators include starting, stopping and moving components, notifying end-users, and changing the state of various devices and applications. Different ubiquitous computing environments may implement these operators differently depending on the architectures and specific characteristics of the environments. However, these low-level implementation details are abstracted away from developers. Hence, developers do not have to worry about how operations are performed in a specific environment and the same program can run in different environments. More details of this programming model are in Chapter 4.

3.5 Composing Activities to form Tasks

The different activities are composed together to create a task. The control flow piecing together the different activities is specified either in a C++ program or in a Lua [19] script. Lua is a powerful, light-weight, interpreted programming language designed for extending applications, and is especially suited for configuration, scripting, and rapid prototyping.

Hence, a task looks like (and has the same expressive power as) any C++ routine or Lua script, except that the variables in the program correspond to classes in the ontology and the functions are activities that have been defined using the Olympus high-level programming model. The parameters of the task are declared in the external parameter XML file. For example, the code segment in Appendix A shows how different activities are composed together in C++ for the slideshow task in Fig 3.1. The lines in **bold** are parameter-obtaining activities; the lines in *italics* are world-altering activities and the underlined lines are state-gathering activities.

The control flow can also be specified in the Lua scripting language. This allows developers and administrators to modify the control flow easily or piece together different activities in the task. For example, Table 4.2 shows such a control flow for the task of migrating an application.

The composite model of a task consisting of several activities borrows from models of workflows like BPEL [12]. BPEL has different kinds of primitive activities like invoking web services or throwing exceptions. These primitive activities can also be combined and executed in sequence, in a loop, or in parallel. However, the key difference between our task model and other workflow models is that our tasks are more dynamic. Since task parameters are explicitly declared in the task parameter XML file, our framework can configure how the task is executed by picking appropriate vales of parameters depending on the current context and the end-user. In BPEL, however, the web services used and the pattern of interaction are pre-specified statically and are difficult to adapt to different situations.

Our task model, like BPEL, has the advantage that developers and administrators can change the control flow or pattern of interaction easily without having to change the underlying components involved (the different activities, services and applications). It also improves the scalability of task development and prototyping. This is because activities and tasks can be composed; portions of tasks and user interactions

can be developed and tested independently.

3.6 Developing a Task

In summary, there are three main steps involved in developing a new task:

1. Decide what are the parameters of the task that would influence execution and describe these parameters in a task parameter XML file

2. Develop world-altering and state-gathering activities or reuse from existing libraries of activities (in C++)

3. Compose a number of these activities (in C++ or in Lua)

3.7 Other Examples of Tasks

In order to give an idea of the kinds of tasks we have developed using the task framework, we give some more examples of tasks that we have developed using it. Most of these tasks are executed in smart conference rooms or offices.

3.7.1 A Lecture Task

The Lecture Task allows a lecturer to specify just the name of the file that he wants to display in a smart conference room or smart classroom. The task execution framework decides all the details of displaying the slideshow including the best devices to use for running the coordinator and the model of the slideshow application, the number of presentations and controllers, and the devices and components to use for presenting and controlling the slideshow. It also starts an attendance application that gets information about the people currently attending the lecture and puts up that information on a public (or private, depending on the policies) display as well.

At the end of the lecture, the lecturer has the option of either terminating the slideshow application or suspending it, so that he can restart the slideshow at the same point later. If he chooses to terminate the application, the slideshow is stopped. If he chooses to suspend it and the state of the application is saved. Later on, the Task Control GUI gives him the option of restarting the same slideshow. When the application is started again, it once again discovers the best configuration for presenting and configuring the slideshow. Hence, if the state or context of the room has changed, the task execution adapts accordingly. For example, a certain plasma screen that was used to display the slides earlier may no longer be available because it is displaying some other information or it is being used for some other purpose, or because it may have failed. In that case, the task execution framework discovers other devices that can be used to display the slides. Figure 3.7.1 shows a flowchart representation of this lecture task. The main point to note about the lecture task is the use of a loop in the flowchart. Essentially, any control construct of C++ or Lua can be used to compose different activities into a task.

3.7.2 A Music Playing Task

The Music Playing task is similar to the slideshow task. The parameters of the music playing task include the name of the music file, the devices to use for the coordinator and model, the number of presentations and controllers and the devices to use to start the presentations and controllers. The presentation of a music playing application is any component that can play a music file like WinAmp or Windows Media Player. One constraint is that the presentation can only play on a device that has a speaker attached to it. The controller of the music playing application is any GUI that allows the user to choose the song from a playlist and adjust the volume and other parameters of the sound being produced.

Just as in the slideshow application, if the task execution framework detects the

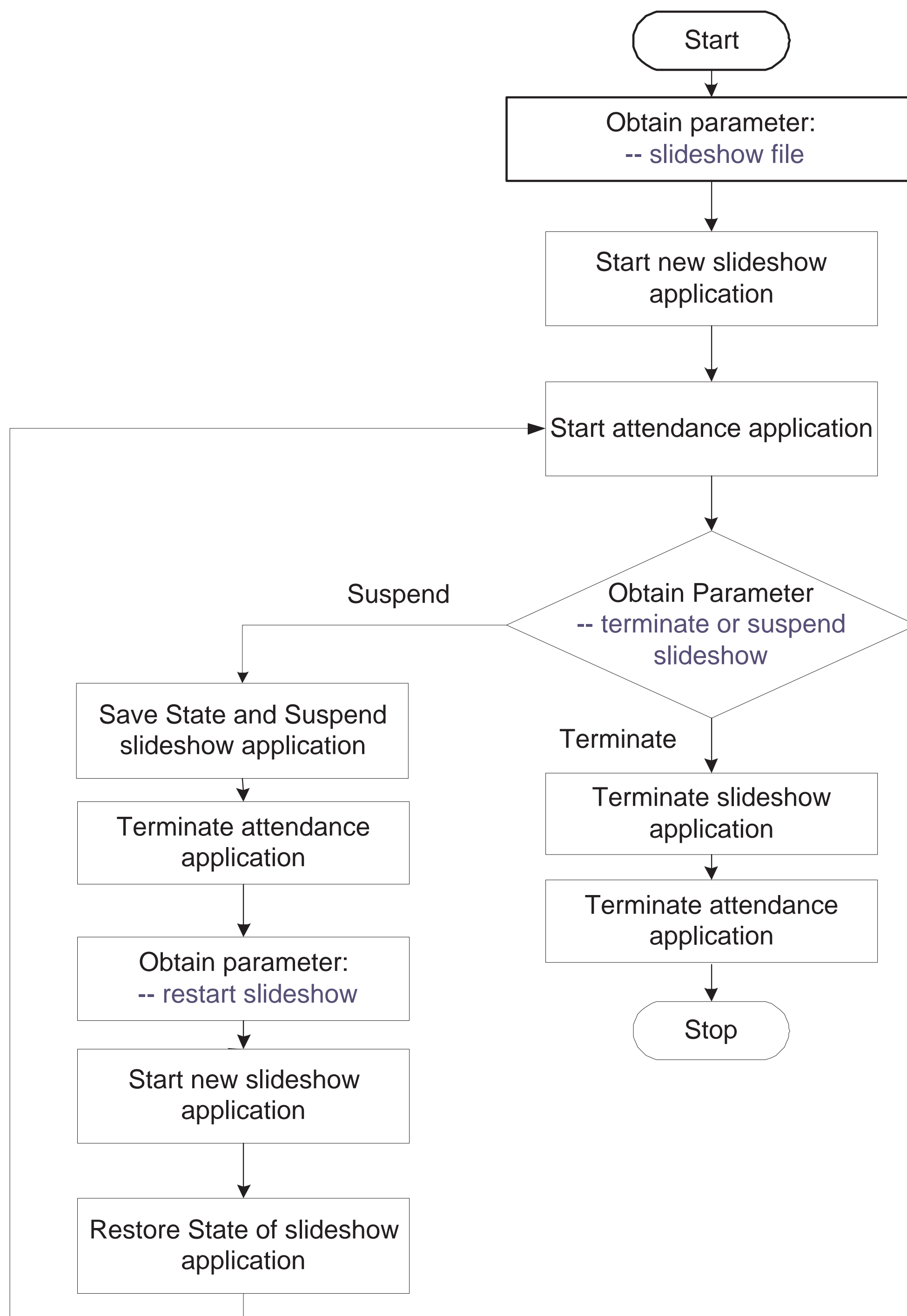

Figure 3.7: Lecture task represented as a flowchart

presence of a music player in the Active Space, it may reuse that application, or else, it may start a new application using appropriate devices and components.

3.7.3 A Collaboration Task

The Collaboration Task allows a group of users to collaborate on a single file. The file may be a text document (in MS Word or other formats), an Excel spreadsheet or an image file. All the users in the collaboration session can then edit this file concurrently using different devices. The users may be in the same Active Space or may be remote. The parameters of the task include the name of the file, the number of users in the collaboration session, and the names of the users. The task execution framework figures out the best application to use (Notepad, Wordpad, MS Word, Excel, Paint, our own Scribble Application, Calculator, etc.) and the best devices for the different users to use. In order to decide the best device, it discovers appropriate devices that are close to the user and that can run the collaboration application. In order to find the location of the users, it uses the Location Service, which is part of the Gaia Location Middleware called MiddleWhere [32]. If it cannot find the location of the user, it asks the user to pick an appropriate device.

3.7.4 A Notification Task

The Notification Task allows a user to send a message to another user. The Task Execution Framework picks the best way of sending the message - using a ticker tape application that runs across different plasma screens in the Active Space, using a speech application called the GaiaSpeechEngine that can do text-to-speech, or using email. The parameters of the task are the name of the user and the message to be sent. The Task Execution Framework picks the best way of sending the message depending on the current state and context of the Active Space where the user is located and the policies of the user and the space. For example, if the current activity in the Active

Space is a meeting, it will not pick the speech application. If the application it decides upon is not running in the Active Space, it also starts the application appropriately. For example, if the ticker tape application is not running in the space, it decides which displays are to be used to start the ticker tape. Similarly, if the GaiaSpeechEngine is not running, it starts it appropriately, taking care that the presentation of this application needs to run on a device that is connected to a speaker.

Chapter 4

Developing Activities using the Olympus Programming Model

Developers program world-altering and state-gathering activities using the Olympus programming model [16]. This programming model allows developers to describe the behavior of a ubiquitous computing program at a high-level, using high-level operands and high-level operators. The associated Olympus framework takes care of the low-level details, including mapping the high-level operands and operators into appropriate concrete operands and operators.

4.1 High-Level Operands

High-level operands in Olympus are Active Space entities including services, applications, devices, physical objects, locations, users and Active Spaces. Each of these basic types of operands is associated with a hierarchy in the ontology and developers can use any subconcept of these basic types while programming. The Olympus framework takes care of resolving these abstract, high-level operands into actual Active Space entities based on constraints specified by the developer, the resources available in the current space, policies and the current context of the space. High-level operands may

be either passed as parameters into the activity or may be declared locally within the activity.

High-level operands used within Olympus programs are also called *virtual entities* (they become concrete entities once the task execution framework discovers suitable actual entities that the high-level operands can be mapped to). Virtual entities that are declared locally are called *local virtual entities* and those that are passed to the program are called *virtual entity parameters*.

Local virtual entities are declared within the program. The class and properties of this entity are also defined within the program. As an example, the developer may want to obtain a device to which he can migrate Bob's suspended slideshow application. This device should have a certain resolution and should be located close to the user Bob. He thus declares three virtual entities: user1 that represents the user Bob, app1 that represents the suspended slideshow application, and a device1 that represents the device with the required properties. He declares user1 to have the name Bob and app1 to be suspended slideshow applications belonging to Bob. He declares that device1 is of class *VisualOutput* (which is a class in the device ontology in Fig 5), is located in the same location as user1 and has a resolution of 800*600. He also declares that if there are many devices that satisfy these constraints then the distance from user1 should be used as the metric, i.e. the one that is closest to user1 should be picked.

```
void migrateSlideshowApp() {
    User user1;
    user1.hasProp("name", "Bob");
    user1.instantiate();

    App app1;
    app1.hasClass("SlideShowApp");
```

```
    app1.hasProp("owner", user1);
    app1.hasProp("status", "suspended");
    app1.instantiate();

    Device device1;
    device1.hasClass("VisualOutput");
    device1.hasProp("location", user1.getProp("location"));
    device1.hasProp("resolution","800*600");
    device1.hasMetric("distance", user1, "ascending");
    device1.instantiate();

    app1.resume(device1);
}
```

The discovery of the concrete user, device and app make use of the Olympus Discovery Service. The instantiate method on the virtual entities triggers a call to the Discovery Service to discover concrete mappings of the virtual entities. For example, in the case of device1, the Discovery Service discovers appropriate concrete classes of devices, including different ontological subclasses of *VisualOutput*. It then discovers instances of these classes that satisfy the different constraints and ranks them in increasing order of distance from the user Bob. Further details of the discovery process are in Sec 6.3 .

In the above program, all virtual entities are declared and used within the migrateSlideshowApp function. However, these virtual entities could also be passed as parameters to the function. Such parameterized functions are used as state-gathering or world-altering activities in the task execution framework. For example, the same migrateSlideshowApp with user1 and device1 being passed as parameters to the function is shown below:

```
void migrateSlideshowApp(User user1, Device device1) {
    App app1;
    app1.hasClass("SlideShowApp");
    app1.hasProp("owner", user1);
    app1.hasProp("status","suspended");
    app1.instantiate();

    app1.resume(device1);
}
```

Such parameterized activities form the basic elements of a task. In the above example, the migrateSlideshowApp activity may be part of a larger task, for example a task that enables a slideshow application to follow a user as he moves around a building. The parameters user1 and device1 are parameters of the slideshowApp entity in the task and may be discovered in a previous parameter-obtaining step. In this case, the class and properties of the parameter are specified externally in the task parameter XML file such as the one in Table 4.1.

The task is then written in C++ or Lua composing the two activities -viz. obtaining the values of the user and device parameters and then invoking the migrateSlideshowApp activity with these parameters. The Lua version is shown below in Table 4.2.

The appAct variable is an over-riding class that defines all the activities as functions. The aqs local variable holds the virtual entity slideshowApp along with its parameters Owner and PresentationDevice. The class and properties definitions of these parameters are taken from the task parameter XML file. The queryForTaskParameters activity invokes the Olympus Discovery Service to discover appropriate concrete instances of the parameters. Finally the migrateSlideshowApp activity is invoked with

```xml
<Entity name="slideshowApp">
  <Parameter>
    <Name>Owner</Name>
    <Class>User</Class>
    <Mode>Automatic</Mode>
    <Property>
      <PropName>name</PropName>
      <PropValue>Bob</PropValue>
    </Property>
  </Parameter>

  <Parameter>
    <Name>PresentationDevice</Name>
    <Class>VisualOutput</Class>
    <Property>
      <PropName>resolution</PropName>
      <PropValue>800*600</PropValue>
    </Property>
    <Property>
      <PropName>location </PropName>
      <PropValue>user1:location</PropValue>
    </Property>
    <Mode>Automatic</Mode>
    <Metric>Distance</Metric>
  </Parameter>
</Entity>
```

Table 4.1: Segment of Parameter XML File for migrating a slideshow

```
appAct = ApplicationActivities:new()


local aqs[]

aqs[0] = {"slideshowApp ", 1, {"Owner","PresentationDevice"}, 2 }

appAct:queryForTaskParameters(1, aqs)


appAct:migrateSlideshowApp(

     appAct:getParamValue("slideshowApp","Owner"),

     appAct:getParamValue("slideshowApp","PresentationDevice"))
```

Table 4.2: Lua script with parameter-obtaining and world-altering activities

the Owner and PresentationDevice parameters just discovered.

The advantages of having the classes and properties of the parameters specified in an external XML file are that it enables rapid prototyping and easy changing of the classes and properties of the parameters. The same migrateSlideshowApp activity can now be reused in a number of different tasks with different properties and values of the parameters.

4.2 High-Level Operators

The other feature of the Olympus programming model are the high-level operators. High-level operators operate on the high level operands (i.e. the different Active Space entities) in different ways. There are, in general, three kinds of high-level operators:

1. *Operators to manage the lifecycle of Active Space entities.* These include operators for starting, stopping, suspending applications and other entities

2. *Operators to query or change the state of Active Space entities.* For example, the state of a light may be on, off or dim. High-level operators allow querying

or changing the state of the light.

3. *Operators to query or change the relationships between Active Space entities.* For example, operators exist to query the relationship between a user and a location (e.g. where is a user?), or for changing the relationship between a device and an application (e.g. move an application to a different device).

4.3 Expressiveness of the Olympus Programming Model

The expressiveness of the programming model depends on the set of operands and operators available. In our Olympus model, we have identified 7 basic types of operands that we found useful in an Active Space : viz. services, applications, devices, physical objects, locations, users and Active Spaces. All these types are associated with hierarchies defined in ontologies.

The set of operators depends on:

- Lifecycle of entities
- State Space of entities
- Relationship Space between entities

4.3.1 Characterizing the lifecycle of entities

For example, the lifecycle of applications and services consists of states like started, stopped, checkpointed and suspended. Devices may be on or off. Users may be unauthenticated or authenticated. Table 7.2 lists some of the life-cycle operators that are implemented in Olympus for our Active Space.

4.3.2 Characterizing the state-space of entities

The state space of an entity depends on the kind of entity. For instance, the state space of slideshow applications consists of the slide number and the name of the file. The state of a user includes the roles he can activate and the set of notification messages that have been delivered to him. Table 7.3 lists some of the state-space operators that are implemented in Olympus for our Active Space.

4.3.3 Characterizing the relationship-space between entities

We also defined a number of operators for managing the relationships. Some of these relationships are captured in Figure 4.2. In order to give an idea of the characteristics of the relationship-space between certain kinds of entities, we describe the relationship-space based on location. The following subsections define some of the relations that are defined between locations, and between a location and any other entity, and also the location-based between two entities.

Relations between two locations

We define several relations between locations or regions based on the Region Connection Calculus (RCC) [49]. RCC is a first order theory of spatial regions. RCC-8 defines various topological relationships: Dis-Connection (DC), External Connection (EC), Partial Overlap (PO), Tangential Proper Part (TPP), Non-Tangential Proper Part (NTPP) and Equality (EQ). Any two regions are related by exactly one of these relations. Figure 4.1 shows some of relationships between locations or regions defined by RCC.

A key relation is that of external connectedness (EC). If two locations or regions are externally connected, it means that it may be possible to go from one region to another. An example of this is two rooms that are connected by a door. However two adjacent rooms that just have a wall (with no door) in between are also externally

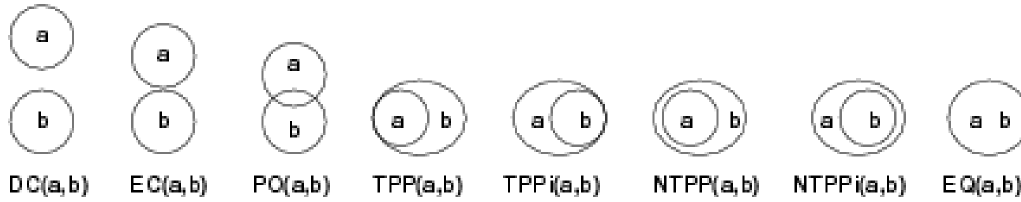

Figure 4.1: Different relations between regions, as defined by RCC

connected. To make this distinction, we define three additional relations:

ECFP(a,b) is true if EC(a,b) and there is a free passage to go from a to b. ECRP(a,b) is true if EC(a,b) and there is a restricted passage to go from a to b. ECNP(a,b) is true if EC(a,b) and there is a no passage to go from a to b.

An example of a restricted passage is a door that is normally locked and which requires either a card swipe or a key to open. The various relations between regions are useful for a number of applications such as route-finding applications.

Another relation between regions is distance. Two kinds of distance measures are used: Euclidean, which is the shortest straight line distance between the centers of the regions, and path-distance, which is the length of a path from the center of one region to the center of the other region.

Relations between an entity and a region

Olympus defines various relations between an object and a region. These relations may be probabilistic if the location of the object is only known with some probability (the Gaia Location Middleware [32] provides location information of mobile entities with a probability that the location is correct). Some of the main relations defined are: (a) Containment: whether an object is within a certain region. (b) Usage: Usage Regions are defined for certain objects (like displays or tables) such that if a person has to use these objects for some purpose, he has to be within the usage region

of the object. (c) Distance: the distance from an object to a region (Euclidean or path-based).

Location-based relations between two entities

The main location-based relations between two objects are (a) Proximity: whether the two objects are closer than a pre-defined distance. (b) Co-location: whether the two objects are located in the same symbolic region (of a specified granularity such as room, floor or building). (c) Distance: the Euclidean or path-based distance between the two objects.

Other relations between any two entities

Besides the relations based on location, Olympus defines a number of other relations between different kinds of operands. Tables 7.4 and 7.5 list some of the relationship-space operators that are defined in Olympus for our Active Spaces. Table 7.6 lists some operators that generate events when some condition on the relationship-space between two operands is satisfied.

4.3.4 Operators for a ubiquitous computing environment

For Active Spaces, we have defined operators that allowed querying, changing or being notified of any change in the lifecycle, state and relationship between entities. Section 7.1.1 evaluates the expressiveness of the model in further detail and also lists and describes some of the operators and operands provided by Olympus.

The set of operators that we have identified is not exhaustive but is sufficient to express many common Active Space behaviors. The set of operators is extensible and is also defined in the ontologies. All Active Spaces that can be programmed using the Olympus model must have an implementation of these operators, though the operators may be implemented differently in different spaces. The Olympus framework

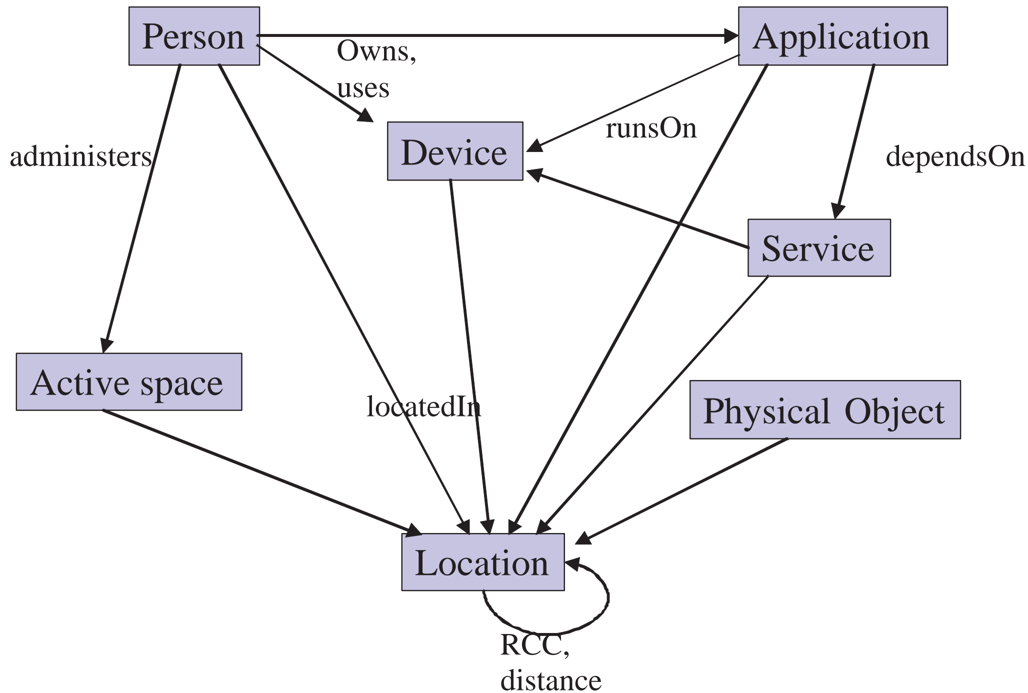

Figure 4.2: Relationships between different kinds of Active Space Entities expressible in Olympus

binds the high-level operators with the appropriate implementation available in the space and and executes the implementation whenever these operators are invoked in a high-level program.

The exact set of operators and operands for any ubiquitous computing environment will depend on the characteristics of the environment. We found the set of operators and operands that we defined useful for many smart space scenarios. One of the broader goals of the Olympus model is to identify different kinds of entities and operators associated with them that may form a part of a minimal standard for pervasive computing environments. Such a standard would allow the same program to run unchanged in different environments.

4.4 Comparison with traditional programming models

Traditional object oriented programming models have the concept of polymorphism, where a base class can be instantiated by any subclass. The Olympus Programming Model has a similar notion, wherein high-level classes (or concepts in the ontology) are mapped to appropriate low-level classes (or concepts) during runtime. The key difference is that Olympus allows intelligent, policy-driven, context-sensitive type coercion. The mapping from high-level classes to low-level classes is done while taking into account the current state and context of the environment, the task being performed, administrator policies and the users' preferences. The other key difference is that the class hierarchy is external to the programming language. The class hierarchy is specified in the ontologies, which are reusable and extensible, and written in a standard format like OWL. Hence the same ontology class hierarchy can be used in different programming languages.

Chapter 5

Why the Task Model? - Exploring the Space of Automation

A key vision of ubiquitous computing is that these environments will eventually ”disappear” into the surroundings and become invisible. In order to become invisible, these environments need to perform tasks in a self-managing and autonomic manner, requiring minimal user intervention. Hence, developers of ubiquitous computing environments need to develop programs or tasks in these environments that are self-managing and autonomic, and that can adapt to the current state and context of the environments. The question, then, is, how self-managing or autonomic should these environments be? In other words, what is the level of automation that must be supported by these environments?

In order to frame the problem more concretely, let us go back to the definition of a task. A task is a set of actions performed by the user or the environment in order to achieve a goal. The level of automation supported by the environment depends on how the set of actions (or the path of actions) that is needed to achieve a goal is obtained. An action can be thought of in terms of an operator acting on one or more operands.

At one end of the automation spectrum is a non-automated, static, imperative

approach where the set of actions required to achieve a goal is completely and precisely specified by a developer. In such an approach, the developer writes a program or a script that contains the exact flow of actions (consisting of concrete operands and operators) required to achieve the goal. At the other end of the spectrum is a dynamic, goal-driven approach, where the set of actions required to achieve a goal is discovered dynamically by the environment. In such an approach, the developer or the end-user) only specifies the goal state to be reached and the environment uses planning or other techniques to come up with a path of actions to get to the goal state.

The totally static approach is not well suited to the needs of dynamic ubiquitous computing environments. The set of actions needed to perform a task or achieve a goal may depend on the current context of the environment, the resources available, the policies of the environment and user preferences. Besides, a static approach where the operands and operators of all actions are hard-coded is not very portable across different environments.

We, thus, investigated the other end of the spectrum where planning was used to help achieve a goal. Developers and end-users could provide abstract goals and a planning framework used a general-purpose STRIPS planner to obtain a sequence of actions to take our prototype smart room pervasive computing environment to an appropriate goal state. Examples of goals in our prototype smart room were displaying presentations and collaborating with local and remote users. Actions in this framework were method invocations on various applications and services and were specified in PDDL [40] files. While this approach worked well in limited scenarios, we found that it did not scale well to larger environments mainly because of the computational complexity of general-purpose planning. The time taken to discover plans increased rapidly with the number of possible actions, as well as with the number of possible values that the parameters of actions could take. For example, introducing an additional device into the planning problem greatly increased the time

needed to discover plans. This is not surprising since general-purpose planning is NP-Complete. Besides, the parameters of some actions were often of infinite range (like strings or integers). PDDL, though, required all possible values that can appear ass parameters to be enumerated. In general, PDDL was far less expressive than IDL, and it took substantial effort to perform abstractions that allowed methods of services and applications to be expressed as PDDL actions. Another drawback was that developers had to specify the pre-conditions and effects of the methods of their services and applications accurately in PDDL files, which was often difficult to do. Since a ubiquitous computing system consisted of a large number of interdependent devices, services and applications, it was difficult for developers to list all the pre-conditions and effects of methods in their own programs. Hence, a totally automated, dynamic approach had serious limitations for applicability in ubiquitous computing systems.

However, we found that most plans generated for a certain goal in our prototype pervasive computing environment consisted of nearly the same set of actions, though with different parameters. In other words, different plans used different devices or different applications to perform the same kind of action. For example, the goal of displaying a presentation produced plans with similar actions involving starting an appropriate slideshow application, dimming the lights and stopping any other applications that produced sound like music players. Different plans just used different devices and applications for displaying the slideshow.

Hence, instead of trying to solve the more difficult problem of discovering a plan of actions to achieve a goal, we decided to use pre-specified, high-level, parameterized plans and discover the best values of the parameters in these plans. This, in essence, is the task model which lies in between the totally static and totally automated ends of the spectrum. In his model, when an end-user describes a goal, our system loads one of the pre-specified plans, discovers the best values of the parameters of this plan

and then executes the plan in a reliable manner. These high-level, parameterized plans are tasks. The parameters of the task influence they way the task is performed. These parameters may be devices, services or applications to use while performing the task or may be strategies or algorithms to use. The advantage of using pre-planned, yet configurable, tasks over discovering plans at runtime is that it is computationally easier and more scalable. It still, however, offers far greater adaptability and portability than non-automated, static programs.

Chapter 6

The Task Execution Framework

6.1 Architecture

Fig 6.1 shows the overall architecture for programming and executing tasks. Developers program tasks with the help of the Olympus high-level programming model [16]. The task programs are sent to a Task Execution Service, which executes the tasks by invoking the appropriate services and applications. The Task Execution Service may interact with end-users to fetch parameter choices and provide feedback regarding the execution of the task. It also fetches possible values of parameters from the Discovery Service. The Ontology Service maintains ontologies defining different kinds of task parameters. The Framework also handles automatic logging and learning. The Logger logs parameter choices made by the user and the system. These logs are used as training data by a SNoW[18] learner to learn user preferences for parameters, both on an individual basis and across different users. A SNoW classifier is then used to figure out user preferences at runtime. The features to be used in the learning process are specified in the learning metadata XML file.

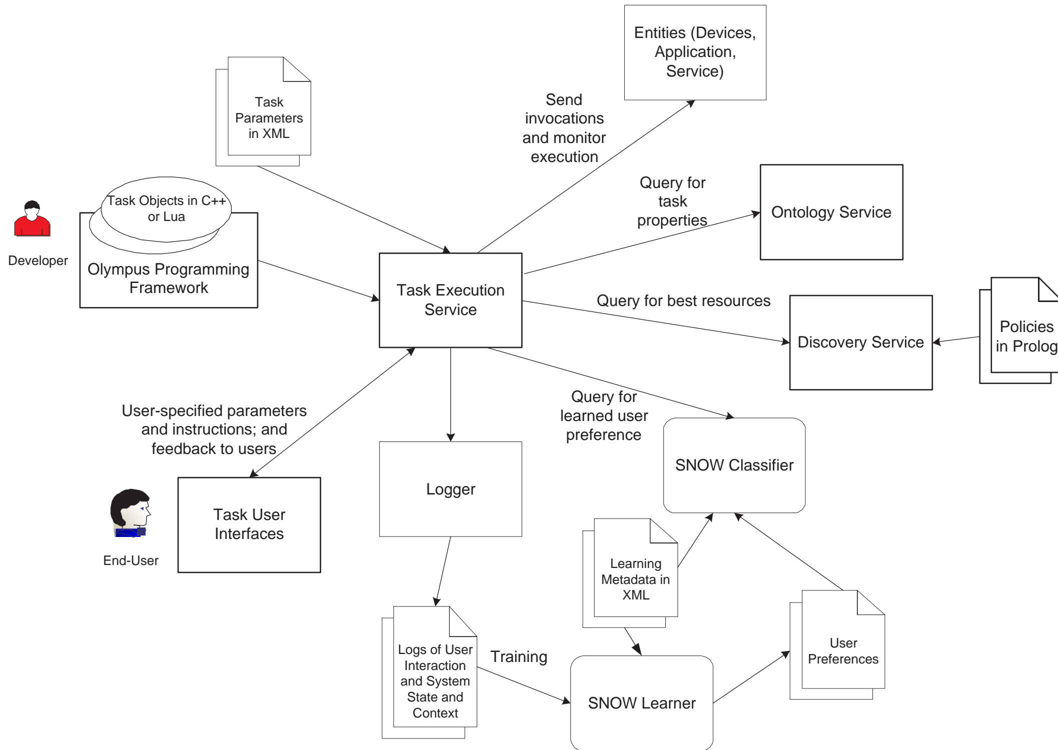

Figure 6.1: Architecture of Task Execution Framework

6.2 Executing a Task

Executing a task involves the following steps:

1. The execution of a task is triggered by an end-user on the Task Control GUI or by any other service in response to an event.

2. The Task Execution Service fetches the task program (coded in C++ or Lua). It also reads the XML file specifying the different task parameters.

3. The Task Execution Service executes the different activities in the task. In the case of world-altering activities, it invokes different applications and services to change their state. In the case of state-gathering activities, it queries the appropriate service to get the required state information. For parameter-obtaining activities, it first queries the Discovery Service for possible values of the parameters. Then, depending on the mode of obtaining the value of the parameter,

it takes one of the following steps:

(a) If the mode of obtaining the parameter value is manual, it presents the end-user with possible values and the end-user chooses one of them.

(b) If the mode of obtaining the parameter value is automatic, it chooses the best value of the parameter that maximizes the utility function metric.

4. The Task Execution Service also monitors the execution of world-altering activities. These activities may use parameter values that have been discovered in a previous parameter-obtaining activity. If the world-altering activity fails due to any reason, the Task Execution Service retries the same activity using an alternative value of the parameter (if there is any).

6.3 Discovering Possible Parameter Values

There are various types of constraints that need to be satisfied while discovering parameter values. These are:

1. Constraints on the value of the parameter specified by the developer in the task parameter XML file.

2. Constraints specified in ontologies

3. Policies specified by an administrator for the current space

The Task Execution Framework uses a semantic discovery process to discover the most appropriate resources that satisfy the above constraints. This semantic discovery process is performed by the Discovery Service.

A key concept employed in the discovery process is the separation of class and instance discovery. This means that in order to choose a suitable entity, the Discovery Service first discovers possible classes of entities that satisfy class-level constraints.

Then, it discovers instances of these classes that satisfy instance-level constraints. Separating class and instance discovery enables a more flexible and powerful discovery process since even entities of classes that are highly different from the class specified by the developer can be discovered and used.

For example, if the task parameter file has a parameter of class “Keyboard-Mouse Input” and that is located in the Room 3105, the Discovery Service first discovers possible classes that can satisfy these constraints. From the device ontologies (Fig 3.5), it discovers that possible classes are Desktops and Laptops. It also discovers that other classes of devices like plasma screens, tablet PCs and PDAs are similar to the required class and can possibly be used in case there are no desktops and laptops in the room. Next, the Discovery Service discovers instances of these classes in Room 3105 and returns these instances as possible values of the parameter.

Fig 6.2 shows an overview of the process of discovering possible values of task parameters. The discovery process involves the following steps:

1. Discovering suitable classes of entities: The Discovery Service queries the Ontology Service for classes of entities that are semantically similar to the class specified by the developer. The semantic similarity of two entities is defined in terms of how close they are to each other in the ontology hierarchy. The Ontology Service returns an ordered list of classes that are semantically similar to the variable class.

2. Checking class-level constraints on the similar classes: The framework filters the similar classes returned by the Ontology Service depending on whether they satisfy class-level constraints specified in the task parameter XML file. These class-level constraints may be specified in ontologies or by the developer. The Jena Description Logic reasoner [22], which is part of the Ontology Service, is used to check the satisfaction of these constraints.

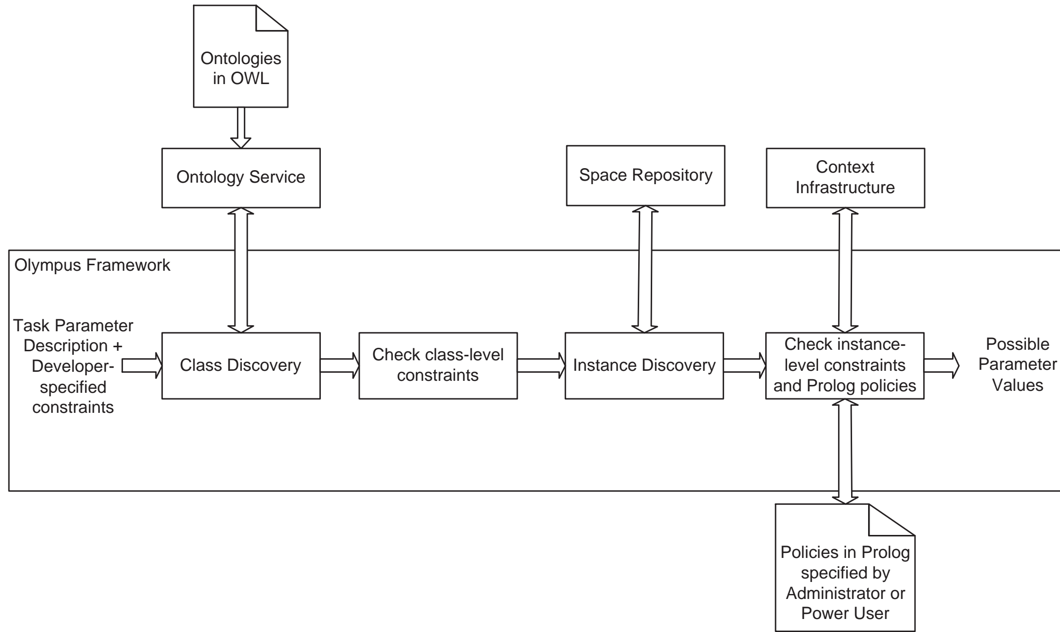

Figure 6.2: Discovery Process

3. Discovering entity instances in the current space: For each remaining class of entity, the framework queries the Space Repository to get instances of the classes that are running in the environment. The Space Repository is a database containing information about all instances of devices, application components, services and users in the environment.

4. Checking instance-level constraints: For each instance returned, the framework checks to see if it satisfies instance-level constraints specified in the parameter XML file These instances are also checked against context-sensitive policies specified in the form of Prolog rules. The final list of instances represents possible values that the task parameter can take.

5. Choosing the best instances: If the best parameter value is to be chosen automatically, the framework ranks the possible instances according to a multi-dimensional utility function and picks the best ones.

The following subsections describe the steps involved in the discovery process in further detail.

6.3.1 Semantic Matching of Classes

In order to allow the discovery of a larger set of classes that can satisfy the developer's requirements, we have developed the notion of semantic similarity of classes. This notion is based on the principle that one entity or resource can be substituted by another if they are semantically similar. For example, in the case of Application Components, the semantics of an application component is based on the tasks it allows the user to perform. So, an application component can be substituted by another component if it allows the user to perform the same tasks in some manner.

For instance, if a developer specifies, in his program, that he needs a PowerPoint View to display slides, then the Olympus framework infers that PowerPoint View can be replaced by an Acrobat Reader view or by a Speech Engine that reads the text in the slides as speech. However, Acrobat Reader is semantically closer to PowerPoint (since it also uses a visual medium and it can also display pictures), and the Speech Engine is a less than perfect substitution. Hence, if it is not possible to display PowerPoint in a certain room (because none of the displays run Windows), then it is better to replace it with Acrobat Reader than with the Speech Engine. However, if the room has no displays or projectors available or if there is a blind person in the audience, then the Speech Engine can be used if there is a speaker in the room.

The process of finding semantically similar concepts makes use of the ontological hierarchy. We implemented an adapted version of the algorithms presented in [34],[33]. In our algorithm, for any two concepts *C1* and *C2*, *C1* matches *C2* with a certain similarity-level based on the following cases:

- If *C1* is equivalent to *C2*, then the similarity-level is 0

- If *C1* is a sub-concept of *C2*, then the similarity-level is 1
- If *C1* is a super-concept of *C2* or *C1* is a sub-concept of a super-concept of *C2* whose intersection with *C2* is satisfiable, then the similarity-level is i+2, where i is the number of nodes in the path in the ontology hierarchy graph from *C2* to the relevant super-concept of *C2*.

The first set includes classes that are effectively the same (but may be described using different terms). For example, the same component may be described as PowerPointViewer in one space and as "PPT" in another space. The ontologies have axioms declaring certain concepts to be equivalent. The second set of classes includes those that are more specific than the query class - i.e. they satisfy all the properties of the query class. The third set includes those classes that are ancestors or children of ancestors of the query class. We just take the leaf nodes, since these are the most concrete classes.

As an example, a query for Presentation components that are semantically similar to PowerPointViewer (in Fig 3.3) gives the following classes:

- Similarity-level 0 : PowerPointViewer (since PowerPointViewer is trivially similar to itself)
- Similarity-level 1: None (since PowerPointViewer has no subclasses)
- Similarity-level 2: AcrobatReader, GhostScript
- Similarity-level 3: WindowsMediaPlayer, Paint, InternetExplorer
- Similarity-level 4: WinAmp, GaiaSpeechEngine

We limit the search to "Presentation" and its subclasses, since we are interested in Presentations only. We, thus, infer that AcrobatReader and GhostScript are closest,

semantically, to PowerPointViewer. Hence, if we find transcoders from the data-formats understood by PowerPointViewer (i.e. ppt files) to the formats understood by one of these two (i.e. pdf or ps files), we could potentially substitute PowerPointViewer by AcrobatReader or GhostScript. The next closest are InternetExplorer, Paint and WindowsMediaPlayer. So, if AcrobatReader and GhostScript are unusable for some reason, we can look for transcoders from ppt to html, an image file or a media file.

The inferring of semantically similar classes of entities that can satisfy developer requirements allows a more flexible and powerful discovery process. It allows the execution of the task to adapt to the resources available in the space, even if they are vastly different from what the developer had in mind.

6.3.2 Checking Prolog Policies

The Prolog policies specify constraints on the classes and instances of entities allowed for performing certain kinds of tasks. An example of a class-level constraint is that no Audio Presentation application component should be used to notify a user in case he is in a meeting. This rule is expressed as:

```
disallow(Presentation, notify, User) :-
        subclass(Presentation,audioPresentation),
        activity(User, meeting).
```

The policies also have access control rules that specify which users are allowed to use a resource in a certain context. For example, the following rule states that a certain application called hdplayer cannot be used by the user for displaying videos if his security role is not that of a presenter.

```
disallow(hdplayer, displayVideo, User):-
        not(role(User, presenter)).
```

The Prolog reasoner, which is part of the Olympus Discovery Service, has a knowledge base that has the current state and the context of the Active Space (such as the machines in it and their characteristics and what components they are currently hosting). The framework uses a predicate model of the Active Space, where the state and context information are represented as predicates, which are asserted into the Prolog knowledge base. This allows the Prolog reasoner to reason about the policies based on the current state and context of the Active Space.

6.3.3 Optimizing Task Execution

Once the Task Execution Service gets possible parameter values from the Discovery Service, it either asks the end-user for the best value on the Task Control GUI or it automatically chooses the best value on its own depending on the mode specified in the task parameter XML file. One of the challenges of ubiquitous computing is that it is very difficult to compare different values since a variety of factors like performance, usability and context come into play. Some of these factors are quantifiable, while others are more subjective and difficult to quantify. In order to get over this problem, the Task Execution Framework employs a multi-dimensional utility function to choose the best value for a task parameter. Different dimensions represent different ways of comparing candidate entities. Some of the dimensions in our current utility function are:

- Distance of the entity from the end-user (e.g. nearer devices may be preferred to farther ones)
- Bandwidth (devices with higher bandwidth may be preferred)

- Processing Speed (faster devices or services are preferred over slower ones)
- Policies specified by the developer or administrator. These policies are written in Prolog and consist of rules that allow inferring the best values of the entities.
- Learned User Preferences. This involves querying a classifier for the best value of a parameter. The classifier is trained on past user behavior.

Entities can have different utilities in different dimensions. A particular entity may be better than others in one dimension, but may be worse in other dimensions. It is often difficult to compare entities across dimensions. Hence, in order to rank all candidate entities for choosing the best one, one of the dimensions must be chosen as the primary one. This primary dimension is the metric for the task parameter.

Depending on the kind of parameter, different metrics may be appropriate for getting the best value. In the case of devices or applications that require direct user interaction, nearer candidate values may be preferred. In other cases, devices or applications may require high bandwidth (such as for tasks based on streaming video) or high compute power (for computationally intensive tasks like graphics rendering or image processing). Prolog policies can be consulted to get the best value of parameters that depend on the current state or context of the environment. Finally, users may have their own preferences for certain kinds of entities- for example, some users prefer using voice-based control of slides while others prefer navigating slides using a handheld device. The actual metric used for comparing different parameter values is specified by the developer in the task parameter XML file. This makes it easy to try different metrics to see which one works for any particular task and situation.

Depending on the metric chosen, different algorithms are used to evaluate the utility function. If the metric specified is distance from the end-user, then the Task Execution Service contacts the Location Service [32] to get the distances from the end-user to the different possible values of the parameter. The Location Service has

access to a spatial database that stores the positions of different static objects (like devices and other physical objects). Besides, various location sensing and tracking technologies like RF badges and biometric sensors are used to detect the location of people and mobile objects. The Task Execution Service gets the distances of different candidate values from the end-user and chooses the closest one.

Performance based metrics like bandwidth and processing speed are evaluated using characteristics of devices. These characteristics are specified in the ontological descriptions of these entities.

The next possible metric is policies. These policies are written in Prolog by an administrator or any other person with expert knowledge on the resources and capabilities of a certain ubiquitous computing environment. The policies specify which parameter values may be preferred depending on the state of different entities, the context and state of the environment, the task being performed, the semantic similarity of the class of the value to the developer-specified class and the end-user performing the task. Policy rules in Prolog assign numerical values to the utility of different entities in different contexts. In case it is difficult to assign numbers, they, instead, specify inequalities between the utilities of different entities. An example of a policy is that high-resolution plasma screens are preferred to tablet PCs for displaying slides in a presentation task:

```
utilityOrder([Device1, Device2], presentation, device, presentationTask, anand) :-
hasClass(Device1, plasmascreen),
hasClass(Device2, laptop).
```

The Discovery Service has access to the Prolog policy files and uses an XSB Prolog reasoner [23] to infer the best parameter value. The Task Execution Service contacts the Discovery Service to get the best value.

User preferences are learned over a period of time by logging user interactions with the system and training a classifier like SNoW over these logs. SNoW (Sparse Network

of Winnows) classifier is a general-purpose multi-class classifier that is specifically tailored for learning in the presence of a large number of features. SNoW learns a target class label as a linear function over the feature space. In our framework, the feature space includes information about the end-user, the task being performed, the state of the environment (such as the devices, applications and services running and their states) and the context of the environment (including other people present, the activity taking place, the locations of the end-user and other people, etc.). The targets that have to be learned are the values of various task parameters.

The utility function is flexible - new dimensions representing other ways of comparing different task parameter values can be added at any time. The automatically chosen best value is suggested to the end-user in the Task Control GUI. The end-user can still modify the chosen value. This allows the end-user to take control of the task execution, in case the framework did not choose an appropriate parameter value

While this utility function is fairly powerful, it does not work as well when the best value depends on a combination of dimensions. For example, the best device to display a presentation may depend both on its distance from the presenter and the size or resolution of the display. We are experimenting with different ways of enhancing the utility function by combining different dimensions.

6.4 Self-Repair

An important characteristic of any autonomic system is self-repair. Actions performed by the Task Execution Service may fail due to a variety of reasons - hardware errors, network faults, software bugs, etc. The Task Execution Service has mechanisms for detecting the failure of actions and recovering from them. Actions performed by the Task Execution Service are in the form of invocations on services or other entities. A failure of an action is detected if the entity on which the method is invoked is not

reachable or does not respond, or from the return value of the invocation (a return value less than 0 indicates failure). A failure can also be inferred by querying another service. For example, if the action is to start a new component on some machine, the Task Execution Service checks to see if the component really started by querying the Space Repository.

Our approach to self-repair is based on the premise that ubiquitous computing environments are device and application-rich. Hence, even if one or more devices or applications fail, there are, normally, alternative devices or applications that can be used perform the same task. Once the Task Execution Service detects the failure of an action, it tries to find alternative values of the parameters involved in that action and retries the action with these different parameters. If the mode of finding the parameter value is manual, it informs the user of the failure of the action and asks him to pick an alternative parameter value from the list of possible values. If the mode is automatic, the framework itself picks the next best value according to the utility function metric specified.

For example, the end-user or the Task Execution Service may pick a certain plasma screen to display a presentation. However, the machine that controls the plasma screen may fail in the meantime, and hence the Task Execution Service cannot start the presentation on it. The Task Execution Service detects this failure and it either prompts the user to pick an alternative device to display the presentation, or it chooses on itself. This alternative device may be another plasma screen, a desktop, a laptop or even a handheld device like a PDA. Our middleware takes care of automatically transcoding the data to an appropriate format that can be displayed by the device.

6.5 The Learning Framework

Machine learning is a common way of developing adaptive systems. Since it is difficult to predict user preferences and behavior, it is not easy to write rules that specify the best values of parameters. Machine learning overcomes this process by deducing the best values of parameters based on past user behavior. The Task Execution Framework includes a learning system that learns user preferences and allows it to customize the execution of the task for different users.

The Learning system includes a SNoW classifier [18] and a Logger. SNoW (Sparse Network of Winnows) is a general-purpose multi-class classifier that is specifically tailored for learning in the presence of a large number of features. SNoW learns a target class label as a linear function over the feature space. In our framework, the feature space includes information about the end-user, the task being performed, the state of the Active Space (such as the devices, applications and services running in the Space and their states) and the context of the Space (including other people present in the Space, the activity taking place in the Space, the locations of the end-user and other people, etc.). The targets that have to be learned are the values of various task parameters. When SNoW is presented a new test example to evaluate, it returns the activations of different parameter values. These activations represent how much the end-user may prefer the value of the parameter in the current context and state of the Active Space. In order to get the best parameter value, the Task Execution Service picks the parameter value with the highest activation.

Whenever the end-user uses the Task Control GUI to perform tasks and select task parameters, the values of the task parameters as well as the current state and context of the environment are logged. The values of the task parameters may have been entered by the end-user or may have been suggested by the framework. If the value was suggested by the framework, the user had the option of keeping the value or selecting an alternative value. In either case, the final value selected is logged

into a training file. The logged values are periodically used as training examples for the SNoW classifier. SNoW learns a weight vector representation of all the target concepts and writes this resultant network into a network file. This network file is then used later for making predictions on new test examples.

In order to configure SNoW easily for learning parameter values for different tasks, the Learning system in our framework stores metadata for the learning process in an XML file. The learning-metadata XML file includes information about the different parameters for the task and the features that are considered to be relevant. The Logger automatically logs the parameter value choices made by end-users on the Task Control GUI along with the features listed in the XML file. The set of parameter values and features can be easily changed for different tasks by modifying the XML file.

The learning metadata XML file has two types of information. First it defines the features that are to be logged during the learning process (Appendix C). Then it defines the possible values of different features and the labels associated with these values (Appendix D).

Appendix C shows the different features that are to be logged and used during learning as specified in the learning metadata XML file. For each feature, the file gives a name (e.g. "Coordinator Device" and gives the type of feature it is (e.g. "Device"). Depending on the value of the feature, the Logger writes a number into the training file. This number lies in the range from the "start number" of the current feature to the "start number" of the next feature. For example, the start number of the feature, Coordinator Device, is 1 and the start number of the next feature, Model Device, is 41. So, labels in the range 1-40 represent different values of the feature "Coordinator Device". In the case of this feature, only one value can be active; so, only one of the labels from 1-40 can be present in the training log file. In the case of other features like Applications Running or Devices Running, multiple values may be active; hence,

many labels may be active in their corresponding ranges.

The possible values of the feature and the labels associated with the different values are specified in Appendix D. For example, Appendix D specifies different possible instance of the type "Device" (e.g. srg-a21p-3-laptop, cs-pc2401-1, etc.) and also specifies the IDs of these devices (0,1,. . .) . So, if the value of the feature named "Coordinator Device" is "cs-pc2401-1", then the active label is got by adding the start number of the feature (1, in this case) with the ID of the device value (1, in this case). Hence, the label 2 is active in this example.

Similarly, the Logger obtains the values of the different features specified in Appendix C and derives the labels corresponding to these values using the ID of the value and the start number of the feature. It then writes the example onto the training file. Appendix E shows an example of such a training file. Each line of the file corresponds to a different training example and a different use of the Task Control GUI to perform a task. This training file is used by the SNoW classifier to obtain a network file, which is used to make predictions.

During the testing phase, the Task Execution Framework uses the SNoW classifier to rank different possible values of a task parameter. Each test example is again constructed by obtaining the current state and context of the environment and deriving the active labels of different features using the specification in the learning metadata XML file (Appendix C and Appendix D). All the target features are then predicted. For example, to predict an appropriate value of the Coordinator Device, the Task Execution Service obtains the activations of different possible values (corresponding to labels 1-4, which represent the different devices) and picks the one with the highest activation.

Chapter 7

Implementation Details and Evaluation

We have implemented the Task Development and Execution Framework on top of our middleware for ubiquitous computing. The various services in the framework such as the Task Execution Service, the Discovery Service and the Ontology Service are implemented as CORBA services. The ontologies were developed using Protégé [15]. A Protégé plugin also offers web-based browsing of ontologies. This allows developers to look up various concepts and properties while developing their programs. We have used the framework to develop and execute different kinds of tasks in different prototype environments in our building. Sample tasks include displaying slideshows, playing music, notifying and communicating with users, and collaboratively working with others on a document, a spreadsheet or a drawing.

The Task Execution Framework uses a number of other components of Gaia to perform various operations. It uses our location middleware, called MiddleWhere [32] to help discover the locations of different static and moving entities and to evaluate different kinds of spatial or location-based relationships between the entities (like those based on RCC [49], distance, proximity, etc.). It uses the Context Middleware [36] for acquiring and reasoning about the current context of the Active Space. The

ontologies and the Ontology Server are part of the Ontology Infrastructure [48] of Gaia.

7.1 Evaluating the expressiveness of the Framework

The main usefulness of any programming framework comes from the variety of behaviors one can program using the features provided by the framework. A programming framework that allows developers to express a large variety of desired behaviors easily is more useful than one that only allows programming a limited set of behaviors. Developing a task involves two main programming stages : programming activities using the Olympus Programming model and composing a number of activities into a task. Hence, we evaluate the expressiveness of the task framework in terms of the expressiveness of the Olympus Programming Model and the Task Model.

7.1.1 Evaluating the expressiveness of the Olympus Programming Model

The Olympus Programming Model essentially provides high-level operators and operands for use in a C++ program. Since C++ is used as the base language, the power and expressiveness of Olympus is equivalent to that of C++, which means that Olympus programs can make use of C++ data types, class structures and other programming features like loops and recursion. In addition, Olympus provides high-level operators and operands that abstract away low-level implementation details from the developer. Section 4.3 describes the expressiveness of the Olympus Programming Model in terms of the space of high-level operators and operands it provides. It, essentially, discusses the expressiveness of Olympus in terms of the set of operands provided, and

the lifecycle, state space and relationship-space of the operands.

In this section, we describe some of the operands and operators provided by Olympus in greater detail. This would help provide further insight into some of the behaviors of an Active Space that can be programmed using Olympus. Table 7.1 lists the set of basic operands currently provided by Olympus. All these basic types are associated with hierarchies specified in ontologies (e.g. in figures 3.3, 3.4 and 3.5). Table 7.2 lists some of the operators that help query or manage the life-cycle of various operands. Table 7.3 lists some of the operators that help query or manage the state-space of various operands. Tables 7.4 and 7.5 list some operators that help query or manage the relationship-space between two operands (i.e. they allow querying or changing the relationship between the target operand and the argument operand of the operator). Table 7.6 lists some operators that generate events when some condition on the relationship-space between two operands is satisfied. These operands and operators are not exhaustive, but help in programming various useful behaviors in our prototype Active Spaces.

7.1.2 Evaluating the expressiveness of the Task Model

The Task Model allows composing a number of activities in order to create a task. The composition can be expressed either in C++ or in Lua. Each activity is a C++ function. Tasks written in C++ are dynamically linked to a library (a dll in Windows) that contains the implementations of the activities in the task. Lua scripts can call C++ functions as well (through the toLua tool [44]). Hence, tasks in Lua can also be linked to the same C++ library containing the implementations of different activities.

Since C++ or Lua is used to express the composition, all the features of C++ or Lua can be used in the composition. Thus, activities can be combined into sequences, decision structures (like if-then-else statements), loops (like for, while and repeat structures) and recursion structures. Activities can also be executed concurrently

Basic Operand Type	Description
ApplicationComponent	Includes the basic components that make up an Active Space application, viz. model, controller, presentation, coordinator and adaptor
Application	Represents an Active Space application as a single entity (which in turn comprises various application components)
Service	Includes various services such as the Naming Service, Space Repository, Olympus Discovery Service and other services useful in the functioning of the Active Space
Device	Includes all kinds of input, output, computational and communication devices
PhysicalObject	Includes all physical objects such as chairs, tables, books, etc.
Location	Includes regions, both indoor and outdoor, and is associated with either coordinate or symbolic information
Person	Includes people of different roles
ActiveSpace	Includes individual Active Spaces (consisting of all the devices, services, applications, users and physical objects within it and associated with a location) as well as hierarchical Super Spaces [45]

Table 7.1: Basic Operands supported by Olympus

Operator	**Target Entity Type (Operand)**	**Behavior**
stop	Service/ Application /ApplicationComponent	Stop target
saveState	Application / ApplicationComponent	Save state of target
suspend	Application / ApplicationComponent	Suspend target
resume	Application / ApplicationComponent	Resume target
on	Device	Turn device on
off	Device	Turn device off
start	Active Space	Start default applications and services in Active Space
stop	Active Space	Stop applications and services in an Active Space

Table 7.2: Some lifecycle operators supported by Olympus

Operator	**Target Entity Type (Operand)**	**Behavior**
name	Service/Application/ ApplicationComponent/ Device/PhysicalObject/Active Space	Get Entity name
queryForTask Parameters	Person	Ask the person to choose a value of one or more task parameters using an appropriate user interface
notify	Person	Send a message to person using an appropriate mechanism
changeFile	Application/ Model	Change the file being displayed or used by the entity - e.g. change the slideshow file of a slideshow application
next	SlideShowModel	Advance a slideshow to the next slide
previous	SlideShowModel	Go back to the previous slide in a slideshow

Table 7.3: Some state-space operators supported by Olympus

Operator	Target Entity Type (Operand)	Argument (Operand)	Behavior
start	Service/Application /ApplicationComponent	Device/ ActiveSpace	Starts component on appropriate device(s) or in space
resume	Service/Application /ApplicationComponent	Device/ ActiveSpace	Resumes suspended component on appropriate device(s) or in space
locatedIn	Service/Application /Application-Component /Device/Person /PhysicalObject/Location /ActiveSpace	Location/ ActiveSpace	Determine containment of target entity in argument entity, or check if target entity is running in or exists in argument location or Active Space
runsOn	Service/Application /ApplicationComponent	Device	Determine if component is running on argument device
migrate	Service/Application /ApplicationComponent	Device/ ActiveSpace	Migrate target entity to argument entity
deployIn	Service/Application /ApplicationComponent	ActiveSpace	Deploy target in a space

Table 7.4: Some relationship-space operators supported by Olympus that operate on the relationship between a target entity operand and the argument operand

Operator	Target Entity Type (Operand)	Argument (Operand)	Behavior
distance	Device/Person /PhysicalObject/Location/ ActiveSpace	Device/Person/ PhysicalObject/ Location/ ActiveSpace	Return distance between target and argument
dependsOn	Application/ ApplicationComponent	Service	Check if target depends on argument
belongsTo	Application/Service/ Device/ PhysicalObject	Person	Check if target is owned by argument
administers	Person	ActiveSpace/ Service	Check if person administers argument
owns	Person	Device	Check if person owns device
uses	Person	Device/ Application/ ApplicationComponent	Check if person uses argument

Table 7.5: Some more relationship-space operators supported by Olympus

Operator	**Target Entity Type (Operand)**	**Argument (Operand)**	**Behavior**
enter	Person/ Device	Location/ ActiveSpace	Generate an event when Person or Device enters argument
exit	Person/ Device	Location/ ActiveSpace	Generate an event when Person or Device exits argument
started	Service/Application /ApplicationComponent	ActiveSpace	Generate an event when target entity is started in Active Space
stopped	Service/Application /ApplicationComponent	ActiveSpace	Generate an event when target entity is stopped in Active Space

Table 7.6: Some event operators supported by Olympus

(e.g. by creating threads).

7.2 Performance

In terms of performance, the overhead imposed by the task execution framework is on average 27% over a static script for the sample tasks that we have developed (displaying a slideshow, playing music, sending a message to a user and starting a collaborative document editing application). This overhead was the time taken to perform the semantic discovery. This included the time taken to access and reason about the ontologies, discover appropriate resources in the Space, check policies and rank the different candidate values of task parameters.

Table 7.7 shows a comparison of the average time taken by the framework for performing a slideshow task and a music task, and the average time it took for static scripts to perform these tasks. The slideshow task (described in Section 3.1) was evaluated in two different configurations - when there was no slideshow application running in the Active Space, and when there was a slideshow application running in the Space. The music task too (described in Section 3.7.2) was evaluated in two different configurations - when there was no music playing application running in the Active Space, and when there was already a music playing application running in the Space.

In these tests, the tasks were configured so that the framework automatically found the values of all parameters. This was done in order to avoid user interaction delays. The static scripts were written in Lua and had a similar structure to the task programs, except that all the values of the task parameters were hardcoded. Thus, the static Lua script did not perform any kind of discovery. Before, the deployment of our task execution framework, most scripts in our Active Spaces were in the form of these static Lua scripts with hard-coded values for the different components and

Task	Static Lua Script Time	Framework Time
Slideshow Task	5.448 sec	6.810 sec
Slideshow Task (with slideshow already running)	2.005 sec	3.876 sec
Music Task	6.509 sec	8.332 sec
Music Task (with music player already running)	3.989 sec	6.309 sec

Table 7.7: Comparison of times taken for tasks in different configurations

devices to use.

The tests were performed in our prototype Smart Room containing 4 plasma screens, a tablet PC and a desktop. Each test was performed 3 times and the average time was taken. All machines ran Windows 2000 and had 1.5 GHz CPUs with 1 GB RAM. The framework time measured was the time taken to execute the whole task program. The static Lua script time was the time taken to execute the whole Lua script.

For both the slideshow and the music task, the tests were done with and without a slideshow and a music application running in the room. If there is already an application running in the room, then the time measured is the time taken to reconfigure the application according to the requirements of the task. If there is not existing application in the room, then the time measured also includes the time to start the application in an appropriate manner.

The task model is very scalable in the sense that tasks can be arbitrarily long. Since each activity is resolved and optimized locally, and there is no global, task-level consistency-checking or optimization, it is possible to compose a large number

of activities in a task, without seriously impacting performance.

7.3 Time Complexity

The overhead imposed by the tasks execution framework is due to the extra time required to discover appropriate values of different task parameters. Hence, we shall now evaluate the complexity of the discovery process. The first step in the discovery process is discovering semantically similar classes of entities from the ontology hierarchy. We normally limit the discovery to classes that are of semantic similarity 4 or less. So, we discover all the leaf classes that are descendants of the 4'th-level ancestors of the given class. The average time to discover all these classes is O(n/i), where n is the number of concepts in the ontology hierarchy and i is the length of the path from the root of the hierarchy to the 4'th level ancestor. In the worst case, that becomes O(n) if the developer specifies a concept that is close to the root of the ontology tree.

The next step is checking class level constraints. The time taken to check the constraints is O(#semantically similar leaf classes) $\times$ O(#class-constraints). The number of semantically similar leaf class is O((n+1)/2) or O(n), where n is the number of concepts defined in the ontology.

The next step is querying the Space Repository or a Prolog Knowledge Base and other repositories to get instances of the similar classes that satisfy instance level constraints. The number of queries sent to the Space Repository or Prolog Knowledge Base is O(n) in the worst case, where n is the number of classes from the previous step, which is of the same order as the number of concepts in the ontology. The Space Repository and the Prolog Knowledge Base have the current state of the environment cached locally, so they can get all the instances in constant time. The number of instances is O(n) $\times$ O(#instances per class). They, then, take O(#instance-constraints)

to check all the instance constraints for each instance.

The final step in the discovery process is ranking the different instances to choose the best ones. Assuming that it takes constant time to get the utility of an instance, this step takes time proportional to O(#instances).

So, the net complexity of the whole process is

$$O(n) + O(n)\times O(\#\text{class-constraints}) + O(\text{instances})\times O(\#\text{instance-constraints}) + O(\#\text{instances})$$

or,

$$O(n) + O(n)\times O(\#\text{class-constraints}) + O(n)\times O(\#\text{instances per class})\times O(\#\text{instance-constraints}) + O(n)\times O(\#\text{instances per class})$$

or,

$$O(n) \times [\ O(\#\text{class-constraints}) + O(\#\text{instances per class})\times O(\#\text{instance-constraints})\ ]$$

where n is the number of concepts in the ontology hierarchy.

7.4 Evaluating task performance by an end-user

Tables 7.8 and 7.9 lists the average times for performing various tasks by an end-user in different scenarios. These times include the user interaction times - i.e. the time it takes for the end-user to interact with the Task Control GUI filling in various parameter values. These times again are the average of 3 separate iterations. In all the tasks, the user had to interact with the task Control GUI on a plasma touch screen using an on-screen keyboard.

The tasks that were performed by the end-user included a slideshow task (described in Section 3.1), a music playing task (described in Section 3.7.2) and a notification task (described in Section 3.7.4). The various tasks were performed under

different configurations of the Active Space and under different configurations of the task itself. For example, in some cases the user had to specify a number of parameters for the task, while in other cases, he only had to specify one parameter value. Also, in some cases, faults were injected into the system by forcefully killing a device or an application. In some cases, certain applications (like a music player or a ticker tape) were already running in the Active Space, while in other cases, these applications had to be started by the framework in an appropriate manner.

The End-User Time measured was the time taken to execute the whole task, starting from the moment when the end-user chose the task he wanted to perform on the Task Control GUI till the time when the task was completely performed. This time included a number of stages such as various user-interaction stages (where the end-user was queried for the best values of some task parameters), some semantic discovery stages (where the task execution framework discovered the possible and the best values of task parameters), some state-gathering activities (where the current state of the Active Space was obtained) and some world-altering activities (where the Active Space was changed in some way - by starting a new application, modifying or controlling existing applications, etc.). In all cases, the end-user performing the task was an experienced end-user, who was familiar with the layout of the room and the applications and interfaces available.

7.5 Other Experiences with the System

We also present some of our initial experiences while using the system. The framework was used to develop activities and tasks for different prototype environments. We found that the framework greatly aided rapid prototyping. Since tasks are programmed at a high-level with the help of reusable, parameterized activities, developers could easily develop new kinds of tasks. They were also able to experiment

Task	End-User Time
Slideshow Task (where user only has to input name of file)	16.549 sec
Slideshow Task (where user specifies all application parameters - devices and classes of all components, name of file)	31.348 sec
Slideshow Task (where user specifies all application parameters and there is an error recovery due to a failed presentation device)	38.871 sec
Slideshow Task (where user specifies all application parameters and there is an existing presentation and the user decides whether to reconfigure it or not)	53.348 sec
Music Task (where user only has to input name of file)	19.418 sec
Music Task (where user only has to input name of file and there is already a music player running)	16.113 sec

Table 7.8: Time taken by user to perform various slideshow and music tasks

Task	End-User Time
Speech Notification Task (where system has to start Gaia Speech Engine in an appropriate manner)	21.321 sec
Speech Notification Task (where Gaia Speech Engine is already running)	10.85 sec
TickerTape Notification Task (where system has to start Ticker Tape application in an appropriate manner)	29.023 sec
TickerTape Notification Task (where Ticker Tape application is already running)	11.251 sec

Table 7.9: Time taken by user to perform various notification tasks

with different configurations of the task and see which worked well by changing the properties of parameters or the mode of obtaining the best values of parameters in the task parameter XML file. The framework also speeded up development time since the discovery of appropriate entities and common operations were abstracted away from the developer.

Tasks developed using this framework were also more portable since they did not rely on specific resources available in or configurations of the environment. Hence, we were able to deploy these tasks rapidly in different prototype environments in our Computer Science building.

One shortcoming of the current deployment of the framework is that most Active Spaces in our Computer Science building are primarily used to give demos. People do not use the infrastructure available in their daily activities often enough. Reasons for this include reliability, usability and the markedly different ways of doing things in multi-device pervasive computing environments as opposed to the single computer

environments that people are more used to. As a result, the usage characteristics of these environments are often skewed to demo scenarios. This, for example, manifests itself in the logs that are used for learning patterns of user behavior. These logs, often, end up reflecting choices made in demo situations rather that user preferences. Hence, it is difficult to evaluate the efficacy and usefulness of the learning framework, although, intuitively, the framework should be able to learn user preferences.

Another limitation of the current system is policy management. We currently have no ways of detecting or resolving conflicts between different policies. Also since the policies are written in the form of Prolog rules, there is no "compile-time" checking of the Prolog rules and it is very difficult to test if the rules actually work without testing them in a real setting. It is also rather difficult to see how different policies may interact with one another to produce new effects. Another problem is that since there is no "compile-time" checking of the Prolog rules, it is very easy to make syntactic errors in the rules that are often only detected at runtime or that may lead to unexpected effects.

Hence, a policy or rule simulator that would allow seeing the behavior of different policies in a certain kind of environment would be very helpful. We are looking at other interfaces for specifying policies as well. A policy checker that can at least detect syntax errors as well as some kinds of logical or semantic errors would be useful. For example, a policy checker can possibly detect if there are two rules that have the same conditions and that lead to conflicting actions to be performed or conflicting facts to be inferred (such as two access control rules that denied and gave access to a certain resource in the same conditions).

In terms of usability, we found that the Task Control GUI allowed end-users to perform common tasks in our smart room. A number of visitors and non-expert users were able to use the Task Control GUI for performing tasks in the environment. This was in contrast to before, when these users needed in-depth instruction on using the

environment. In particular, the framework helped reduce the prior knowledge about the environment required by end-users to perform tasks. The framework also reduced the number of actions required to be performed by users for configuring the space, especially in the case of failures.

Since tasks are broadly specified as well-structured flowcharts, they are especially useful when there is a well defined sequence of actions that the user and the system can take to achieve his goals. This model, however, does not allow spontaneous or ad-hoc interactions, where the end-user does not have a clearly defined goal and wants to experiment or try different things.

Chapter 8

Development Complexity Analysis

A key purpose of the framework is to reduce the complexity of ubiquitous computing systems. However, there are no commonly used metrics for measuring the complexity of ubiquitous computing systems, or for any large-scale distributed systems, for that matter. Hence, in order to measure how well our system does in reducing system complexity, we define various aspects of system complexity and propose ways of measuring them. These measures are based on widely used software engineering metrics. We then show how our framework has reduced the complexity of developing applications based on these measures.

8.1 Measuring Development Complexity

The size and complexity of distributed computing systems have been increasing inexorably in the recent past. Large-scale distributed systems such as internet systems, ubiquitous computing environments, grid systems, storage systems, enterprise systems and sensor networks often contain immense numbers of heterogeneous and mobile nodes. These systems are highly dynamic and fault-prone as well. As a result, developers find it difficult to program new applications and services for these systems; administrators find it difficult to manage and configure these complex, device-rich

systems; and end-users find it difficult to use these systems to perform tasks.

System complexity has been widely identified to be an important problem [14],[39]. However, the term ”complexity” is often used loosely. There are no standard definitions of complexity or ways of measuring the complexity of large systems. As with so many complex things, complexity means different things to different people.

The systems research community has been actively looking at different approaches to reduce the complexity of systems. These approaches often take the form of middleware or programming frameworks to simplify the task of developers; various system management tools for administrators; and intuitive user interfaces for end-users. However, today, there is no way of formally, and quantitatively, saying that a certain solution does reduce complexity, or that one solution is better than another. We hope that our proposed aspects and metrics of complexity will allow people to compare solutions in a more scientific way, as well as guide future solutions to tackling the problem of complexity.

In general, system complexity can be described as a measure of how understandable a system is or how difficult it is to perform tasks in a system. A system with high complexity requires great mental or cognitive effort to comprehend, while a system with low complexity is easily understood. In this thesis, we identify five aspects of distributed system complexity: Task-Structure Complexity, Unpredictability, Size Complexity, Chaotic Complexity and Algorithmic Complexity (Fig 8.1). We describe the causes of these different aspects and propose ways of measuring them. We also show how these aspects of complexity impact different classes of people - developers, administrators and end-users.

8.1.1 Task-Structure Complexity

Task-Structure Complexity measures how difficult it is to understand how to perform a task in a distributed system. This complexity aspect takes a graph or flowchart

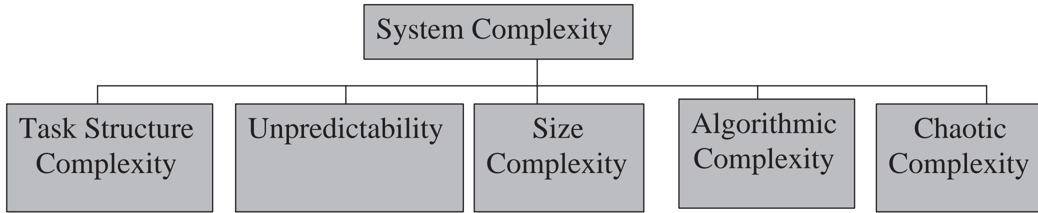

Figure 8.1: Different Aspects of System Complexity

representation of a task and gives a measure of how complex the structure of this graph is. For developers, the task graph represents the structure of the program. For administrators and end-users, the task graph represents the structure of the set of actions that they need to perform to achieve a goal. In order to measure task-structure complexity, we extend a metric from software engineering called cyclomatic complexity[8]. Cyclomatic complexity measures the number of linearly independent paths through the task graph; i.e. it gives the number of possible ways of executing the task. The formula for cyclomatic complexity (CC) is: CC = E - N + p
where E = the number of edges of the task graph
N = the number of nodes of the task graph
p = the number of connected components

The term 'p' is normally equal to 1 for a single process. 'p' may be more than 1 if several concurrent processes need to be undertaken to perform a task.

The other assumption in this formula is that there is only a single end goal state. So, if there are branches in the task graph, all branches finally merge into a single end goal state.

The cyclomatic complexity gives a measure of the number of decision points in the program. Decision points are those from where the task execution can proceed in different directions. Common decision points in distributed systems arise from choosing between multiple ways of performing the task and recovering from failures. A task with a number of decision points (such as one with a number of branches and loops) has a larger cyclomatic complexity than a task that follows a linear sequence

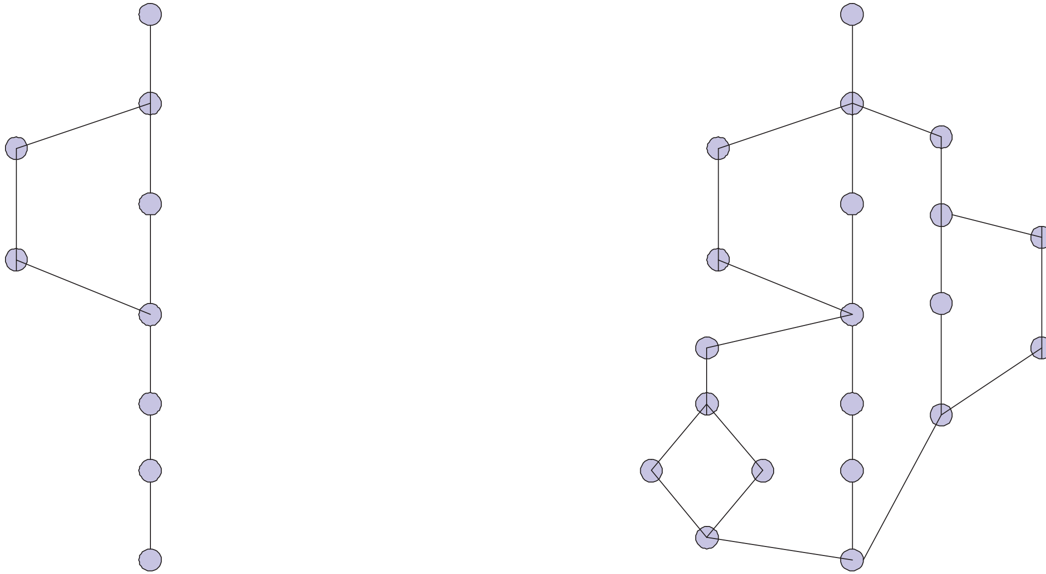

Figure 8.2: Task graphs with different task-structure or cyclomatic complexities

of steps. For example, in Fig 8.2, the task graph at the left has a lower cyclomatic complexity than the one at the right. The task graph at the left has a cyclomatic complexity of CC = 9 - 9 + 1 = 1. The task graph at the right has a cyclomatic complexity of CC = 25 - 20 + 1 = 6

For developers, if the program they are writing has a high cyclomatic complexity, it requires greater effort for developing, testing and maintaining it. For each decision point, developers have to be aware of the various choices available and describe how the program should proceed for each choice. Cyclomatic complexity is also related to the number of test cases that have to be written for testing a program.

For administrators and end-users, a task with high structural or cyclomatic complexity requires more cognitive effort to understand and perform. Administrators and end-users may not be aware of the different choices available at decision points or what is the best choice for the current state of the distributed system and task at hand.

8.1.2 Unpredictability

Unpredictability gives a measure of how difficult it is to predict the effects of an action in a distributed system. An important element that affects predictability is the amount of randomness or entropy in the system. The higher the entropy of the system, the more difficult it is to predict the state the system is in after performing an action. The entropy of a system is measured using the probability distribution of the possible states of the system. If on performing an action, the system is in one of k different states with probabilities $p1, p2, ...pk$, then the entropy of the system, H, is:

$$H = \sum_{i=1}^{k} p_i log_2 \left(\frac{1}{p_i}\right)$$

The term $log_2 \left(\frac{1}{p_i}\right)$ is often called the surprisal factor. Surprisal is the degree to which one is surprised to see a result. If the probability of an event is 1, there is zero surprise at seeing the result. As the probability gets smaller and smaller, the surprisal goes up. Hence, if the system may only be in a small number of states, each with relatively high probability, then the entropy is low and one is unlikely to be surprised very often. But if the system can be in a large number of rare states, then the entropy or unpredictability is high.

Unpredictability in distributed systems often results from dynamism, failures and race conditions. If an action performed by a service or application is unpredictable, then it becomes difficult to test and maintain it for developers. Automation may also create problems of unpredictability for end-users.

There are two ways in which unpredictability of a system can be reduced. One is by reducing the number of states that the system can be in. The other is by increasing the probability of a few ”desired” states and reducing the probability of other states.

8.1.3 Size Complexity

Another measure of system complexity is the size of the distributed system. Traditionally, the size of a distributed system is measured by the number of nodes, devices, services, applications or other components. In addition, a distributed system may have high cognitive complexity if users need to be aware of a large number of concepts in order to use the system. A concept is any logical item of knowledge defined or used by the system. A concept includes abstract notions like file-types, security policies, context information, device characteristics and QoS parameters. A large number of concepts contributes to greater difficulty in understanding the system as a whole. Hence, the size of the body of knowledge required to develop applications for the system, manage the system or use the system to perform tasks is an important measure of complexity.

8.1.4 Chaotic Complexity

Chaotic Complexity refers to the property of systems by which small variations in a certain part of the system can have large effects on overall system behavior. Chaotic complexity makes it difficult to understand systems. It often results from a lack of modular design and from a number of inter-dependencies between different parts of the distributed system.

As an example, policies are often a source of chaos in a system. Policies, such as access control policies, often have the power to affect different parts of the system, especially because many distributed systems do not have ways of checking the consistency of different policies. Hence, it is often fairly easy to write policies that cause unexpected behaviors. For example, it may be easy to write a policy that denies access to all resources for all people, accidentally.

An important factor contributing to chaotic complexity is coupling between different components. There are different kinds of couplings[55]. Some of these couplings

(in order of increasing complexity) are:

1. Components are data coupled if they pass data through scalar or array parameters.
2. Components are control coupled if one passes a value that is used to control the internal logic of the other.
3. Components are common coupled if they refer to the same global data.
4. Components are content coupled if they access and change each other's internal data state or procedural state.

There are many reasons why low coupling between components or modules is desirable [54]. Fewer interconnections between components reduce the chance that a fault in one component will cause a failure in other components. Also, fewer interconnections reduce the chance that changes in one component will affect other components, which enhances reusability. Finally, fewer interconnections reduce administrator and programmer time in understanding the details of the system.

One way of measuring coupling between components is fan in - fan out complexity [53]. This measure maintains a count of the number of data flows into and out of a component plus the number of global data structures that the component updates. The fan in - fan out complexity of a component is given by the formula: $Complexity = Length \times (Fan\text{-}in \times Fan\text{-}out)^2$.

Length is any measure of the length of a program such as lines of code.

However, coupling is just one of the factors that could lead to chaos. We are still in the process of investigating other factors that could cause small variations in one part of the system to lead to large variations in system behavior.

There is a subtle difference between chaotic complexity and unpredictability. Chaos is deterministic - i.e. it is possible to say, deterministically, what would be the

overall change in system behavior on changing any part of the system, given enough knowledge about the architecture and functioning of the system. Unpredictability, however, is intrinsically non-deterministic (or probabilistic). It covers aspects that cannot be predicted deterministically, and that may occur due to random errors or race conditions, or due to insufficient knowledge about the workings of the system.

8.1.5 Algorithmic Complexity

The traditional definition of the complexity of an algorithm is in terms of its time and space requirements or its relation to Turing machines or universal computers. However, there is also a cognitive aspect to algorithmic complexity, which is the effort required to understand an algorithm. There is often a trade-off between the performance and cognitive aspects of algorithmic complexity. Simple algorithms are often brute-force in nature and may have high space or time complexity. However, more sophisticated algorithms that reduce the space or time complexity have a high cognitive complexity. A simple example is the use of an $O(n^2)$ algorithm for sorting (like insertion-sort or bubble-sort) as opposed to an $O(nlogn)$ algorithm (like quick-sort).

The cognitive algorithmic complexity can be measured using Halstead's measures [17]. These measures principally estimate the programming effort, but can be extended to measure the effort required to understand an algorithm. The Halstead measures are based on four scalar numbers derived directly from a program's source code:

n_1 = the number of distinct operators

n_2 = the number of distinct operands

N_1 = the total number of operators

N_2 = the total number of operands

From these numbers, various complexity measures are derived:

Program length, $N = N_1 + N_2$

Program vocabulary, $n = n_1 + n_2$

Program Volume, $V = N \times (log_2 n)$. The program volume measures the information content of a program or the size of implementation of an algorithm.

Difficulty, $D = (n_1/2) \times (N_2/n_2)$. The difficulty of a program is also related to the error-proneness of the program.

Effort, $E = D \times V$. The effort to implement or understand a program is proportional to the volume and to the difficulty level of the program.

This measure of complexity is more relevant for programmers and administrators who write programs and scripts for performing different kinds of tasks.

Halstead's measures are related to the amount of information contained in a program. Another metric that measures the information content of any object is Kolmogorov complexity [46]. Kolmogorov complexity is the minimum number of bits into which a string can be compressed without losing information. This is defined with respect to a fixed, but universal decompression scheme, given by a universal Turing machine. Another way of looking at it is that the Kolmogorov complexity of an object is the length of the shortest computer program that can reproduce the object.

It is a bit more difficult to measure the Kolmogorov complexity of any program since one has to devise a Turing Machine that can generate this program. Also, in general, the Kolmogorov complexity is not computable. The Halstead's measures, however, offer an easier way of measuring the information content of a program.

An important point about Halstead's measures is that in order to measure them, one has to decide what constitutes the set of operators and operands in a program. Commonly occurring blocks of code (like iterating through the elements of a list) may be considered as a single operator (iteration) and a single operand (the list),

although the actual bloc of code may have many more operators (such as the counter or iterator) and operands (such as incrementing the counter, checking for end of list, etc.). Thus, the choice of the set of operators and operands may be made depending on the skill of the programmer.

8.1.6 Why these metrics?

There has not been much work in studying the intrinsic complexity of using distributed systems from the point of view of developers, administrators and end-users. The main reason why we need a different set of metrics or a benchmark to measure the complexity of systems is that most existing measures are either inadequate or flawed.

In the case of developers, the usefulness of a middleware, a design pattern or a programming framework is often measured in terms of metrics like the lines of code or the development time required for programming a certain service or application. For example, many publications that propose new middleware, programming frameworks, development environments or languages evaluate their systems in terms of lines of code and development time (as man-hours or man-months). While these metrics are useful, they have a number of limitations since they depend heavily on coding language, coding styles and developer skill. Also, they do not capture the intrinsic difficulty or complexity of developing or maintaining code. This intrinsic difficulty or complexity of systems arises from the complexity of the problem domain, the state-space explosion of large systems, the non-continuous behavior of discrete systems, the non-linear scaling properties of most systems and the unpredictable influence of external events. Our view is that the complexity metrics we have proposed capture the intrinsic difficulty of building services and applications better than the standard metrics. Our complexity metrics, together with the existing standard metrics, would help in providing a better picture of the usefulness, maintainability and complexity of a distributed system or a middleware.

Also, some of our metrics have been used in the software engineering field for a while. Hence, there is a common consensus on what values of some of the metrics like Cyclomatic Complexity and Halstead's Measures are appropriate to promote understandability, programmability and maintainability of programs. For example, the Halstead volume (V) of a function is recommended to be between 20 and 1000. Volumes greater than 1000 means function is doing too much. The Volume of a file should be between 100 and 8000. Also, the Cyclomatic Complexity of a function is recommended to be less than 15.

The other aspects, including unpredictability and cognitive size complexity, represent important characteristics that make systems difficult to program and use. However, there are no well-accepted metrics for evaluating these aspects. Hence, we propose our own metrics, including entropy and number of concepts, to measure these aspects of complexity. So, although there are no standard guidelines to define appropriate values for these metrics, one can still compare different systems using these measures.

For administrators and end-users, too, there are no standard ways of measuring the complexity of managing and using systems. One metric that is sometimes used in HCI is GOMS [52], which allows calculating the time it takes for achieving a goal with a certain user interface. However, this metric does not really capture the complexity of using the interface. As we have described earlier, many of the aspects that cause complexity for developers also cause complexity for administrators and end-users. Hence, it is possible to extend the software engineering-based and other metrics for administrators and end-users as well.

User studies are also often undertaken to evaluate interfaces for end-users, and in some cases, administrators. These user studies either measure the time or effort taken to perform a task using an interface, and the failure or error rate while performing the task (i.e. the number of times the user made a mistake while using the interface).

Again, such studies heavily depend on the characteristics and skill-sets of the users. Besides, they may or may not say anything about the intrinsic understandability or complexity of using a system. Finally, it is difficult and expensive to conduct fair user studies, and in some cases it may not be possible to conduct user studies at all. In such cases, it would be useful to have empirical complexity metrics, such as the ones we have proposed, in order to evaluate the system. The complexity metrics would also allow evaluating a system and its interfaces at an early stage and help influence design decisions. We believe that a combination of user studies and our complexity metrics can help provide a more complete picture of the complexity of using a system to perform tasks.

A key advantage of the various complexity metrics is that they help in comparing the complexities of existing distributed systems, both quantitatively and qualitatively. Besides, they also allow evaluating new middleware, programming models and interfaces in terms of the reduction in complexity they may offer to developers, administrators and end-users. This would help system designers to argue for or against the use of specific solutions for various kinds of tasks.

8.2 Features of the Task Execution Framework that help reduce Complexity

The task framework has a number of features that help reduce some of the aspects of complexity that we just defined. Three key features that it has to reduce complexity are:

- Self-configuration and self-repair to reduce task-structure complexity and unpredictability
- High-level programming to reduce algorithmic complexity and unpredictability

- Use of ontologies to enable semantic interoperability and to reduce cognitive size complexity

8.2.1 Self-Configuration and Self-Repair

Self-configuration and self-repair help reduce task structure complexity. Self-configuration allows the specification of tasks to be more linear, since the middleware takes care of making choices at various decision points. Hence, developers do not have to worry about choosing appropriate values at decision points. Self-repair allows programmers to not have to worry about dealing with certain kinds of failures. In distributed computing, many of the decision points that contribute to task-structure or cyclomatic complexity arise from checking for exceptions and failures. Hence, self-repair enables eliminating many decision points in programs.

Self-repair also reduces unpredictability since it increases the probability of a few desired states corresponding to the successful execution of a program or successful performance of a task, while reducing the probability of failure.

Our framework achieves self-configuration and self-repair with the help of the parameterized task model. Developers program the ubiquitous computing environment using tasks or workflows and specify parameters that influence how exactly the task is executed. The framework achieves self-configuration by automatically or semi-automatically picking appropriate values of task parameters, and self-repair by picking alternative values of parameters in case the initial choice fails.

8.2.2 High-Level Programming

High-level programming involves the use of high-level operators and high-level operands in programs. This helps reduce cognitive algorithmic complexity by reducing the number of operators and operands required to be used to achieve a certain goal in a program. This, in turn, reduces the program volume and programming effort. High-

level programming, along with self-configuration and self-repair, can also help reduce unpredictability. Since developers specify their programs at a high-level, this gives the middleware more flexibility in choosing an appropriate way of executing the program and recovering from failures. Thus, the unpredictability of program execution, arising from dynamism and failures, is abstracted away from developers. Besides, high-level programming also reduces the number of states visible to developers and end-users, and thus reduces entropy by reducing the number of possible states the system can be in.

Our framework is built on top of a high-level programming model. Tasks in our framework consist of a set of smaller sub-tasks called activities. An activity may change the state of the environment by performing an action or it may query appropriate services to get the current state or context. Activities are programmed with the help of a Olympus high-level programming model [16] that allows developers to program in terms of high-level operators and operands. The middleware takes care of the low-level details, including mapping the high-level operands and operators into appropriate concrete operators and operands.

8.2.3 Ontologies for Organization of Concepts and Defining Semantics

The use of ontologies tackles cognitive size complexity by using a divide and conquers approach. Ontologies are used to define hierarchies of concepts used in the system. Ontologies are a standard way of representing domain knowledge and semantic information about concepts in a reusable manner. They allow different parties to become aware of the various concepts used in the system and the relationships between these concepts. Defining hierarchies of concepts also helps developers and end-users in understanding the details of the system at a high level, while allowing them to drill down to specific details if they want to do so.

Our framework uses ontologies to define hierarchies of different services, devices, applications, contexts, data types and other concepts. Ontologies also define the different high-level operators and operands that are used by the Olympus programming model; hence a high-level program can run in any system that follows the same ontology (or that uses an ontology that has been mapped to the ontology followed by the high-level program). Developers, administrators and end-users use the concepts defined in the ontology while developing applications and configuring the system. Ontologies organize the knowledge space into different segments and thus, help reduce the number of concepts one needs to know to program a task.

8.3 Evaluating the Reduction in Complexity

We now evaluate the reduction in complexity that the framework provides for three different kinds of tasks: a slideshow task, a messaging or notification task and a collaboration task. The slideshow task is similar to the running example in this paper (described in Section 3.1), where the developer writes a high-level script (corresponding to the flowchart in Fig 3.1) and the framework automatically or semi-automatically (i.e. with the help of the end-user) deduces the best values of various parameters like the devices to start different components in and the classes of the components for displaying the slides and controlling the slideshow. The notification task (described in Section 3.7.4) allows an end-user to write a message for any user in the system, and the framework automatically or semi-automatically figures out the best way sending the message - using email, a ticker-tape across one or more displays in the room in which the user is in or using a speech output. The collaboration task (which was described in Section 3.7.3) allows the end-user to specify end-users who should be involved in a collaboration session and the shared file which all users work together on (the shared file can be a text file, a word document, an image file or an excel

spreadsheet). The framework automatically or semi-automatically chooses the best device and application for each user to use for collaborating.

In the case of each of the above tasks, we compared the complexity of developing the tasks with and without using the framework. Hence, the following evaluations compare two tasks that have equivalent behaviors and functionalities, one of which makes use of the facilities provided by the framework, and the other which just makes use of the facilities that Gaia provides minus the framework. Hence the second kind of task, which did not make use of the framework, required many of the self-configuration, self-repair and policy checking features to be specified within the program.

Table 8.1 shows the task-structure complexity and cognitive algorithmic complexity of the three different kinds of tasks without and without using the framework in a certain configuration of our prototype pervasive computing environment. For example, the task-structure complexity of the task graph of a slideshow task is 2 when the framework is used; while an equivalent graph when no self-configuration is done has a task-structure complexity of 1192 (assuming there are 10 devices in the room that can be used to host different presentation and controller components). The cyclomatic complexity of a function is normally recommended to be less than 15 for enabling understandability. When no self-configuration is done, the task-structure complexity is high because there are many different ways of configuring the slideshow with different devices and components. Also, various policies have to be checked to choose the best way of configuring the task and the execution has to be monitored for failures.

The Halstead's measures for cognitive algorithmic complexity of the slideshow task while using the framework are: $n_1 = 4$, $n_2 = 9$, $N_1 = 7$ and $N_2 = 18$. This gives a volume of 92.51, a difficulty of 5.14 and effort of 475.5 . An equivalent program without using the framework had $n_1 = 17$, $n_2 = 32$, $N_1 = 145$ and $N_2 = 266$, giving a

	TSC -Fwk	TSC +Fwk	CAC -Fwk	CAC +Fwk
Slideshow Task	1192	2	70.66	5.14
Notification Task	484	0	34.62	1
Collaboration Task	476	1	38.92	3.07

Table 8.1: Comparison of Task-Structure Complexity (TSC) complexity and Cognitive Algorithmic Complexity (CAC) with and without support of our framework

volume of 2307.65, a difficulty of 70.66 and an effort of 163058.55 . The main reason for the difference in difficulty and effort is that the high-level operators provided by Olympus, along with the self-configuration and self-repair features of the framework, allows developers to focus on the core logic of the tasks and abstracts the low-level details away. The V (volume) of a function is normally recommended to be between 20 and 1000 to enable easy programmability and understandability.

The framework's use of ontologies also helps reduce the cognitive size complexity. For example, a developer can just specify that a VisualOutput device should be used to display a slideshow. He does not have to know all the different kinds of devices (including the various subclasses of VisualOutput device). He also does not have to care about the actual concrete instances of these subclasses or their properties. The framework takes care of choosing the appropriate subclass and instance of VisualOutput device.

Finally, the high-level programming model also helps reduce the unpredictability aspect of complexity. For example, the startApplication operator provided by Olympus automatically decides the best devices and components to use for starting a certain kind of application, recovers from failures and has transactional semantics from the point of view of the developer. If the operator is successful, an application is started, and if it fails, the system is restored to its initial state. Hence there are just 2 possible next states after this operator is invoked. On the other hand, if our

framework was not present, the developer has to worry about several intermediate states of starting the application and recover from failures as well. For example, different components that make up the application may fail, or the devices on which components are started may leave the environment. Hence, the entropy of the system from the point of view of the developer is far greater than when he uses the high-level programming model.

8.4 The Complexity-Flexibility Trade-off

An important effect of reducing complexity through high-level programming using Olympus and the task development framework is that the flexibility of the developer is also reduced. While high-level and task-oriented programming abstracts away many of the low-level details, it is also not as expressive or flexible as lower-level programming and does not allow developers to perform certain kinds of operations. Flexibility is related to the number of different states that a language or system allows to be reached, or the number of different transitions between states offered by operators of the language or system.

The complexity-flexibility trade-off plays out at other levels of programming as well. For instance, machine-level or assembly-level programming is probably the most expressive and flexible since it allows developers to do pretty much anything allowed by the instruction set of the processor. However, they are also incredibly complex to program in. In particular, the cognitive algorithmic complexity of a program in machine-level or assembly-level is very high. Higher-level programming languages like Java are less complex, but also allow lesser flexibility. For example, developers cannot cause arrays or buffers to overflow, or write data to arbitrary memory locations. Also, applets may not open sockets to arbitrary hosts or access the local filesystem.

Fig 8.3 shows a possible graph representing the tradeoff between flexibility and

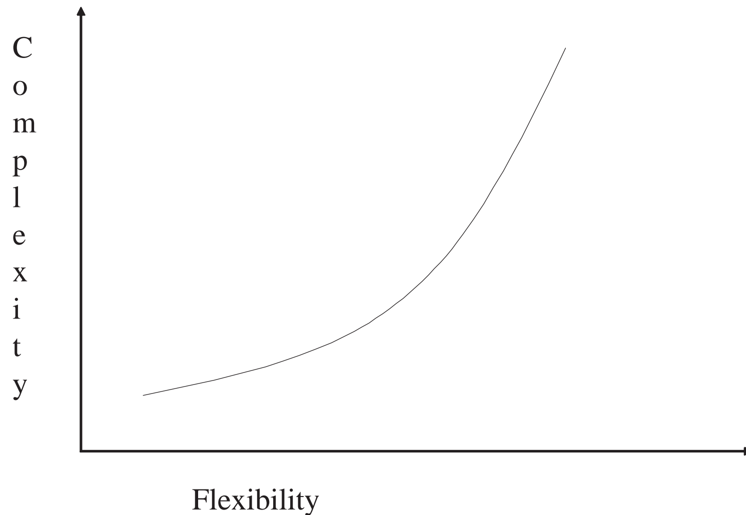

Figure 8.3: The Complexity-Flexibility Tradeoff

complexity. The exact shape of the curve depends on the units used to measure flexibility and complexity, but, in general, as the flexibility and expressiveness of a programming model increases, the complexity of the model also increases. As the complexity of a programming model and the programs in it reduces, the flexibility and expressiveness of the programs also reduces. Hence, depending on the needs and requirements of the developer, an appropriate flexibility-complexity point must be chosen. The choice needs to be made depending on how finely the developer wants to program the system and the level of control and abstraction he desires.

8.5 Distinguishing between the complexity of a system and the complexity of using a system

There is an important difference between the intrinsic complexity of a system and the complexity of using a system to perform tasks. While a system can be very complex, internally, it may still be easy to use for performing various kinds of tasks. For example, cars are extremely complex systems internally; however, the interface exported by a car to end-users (or drivers) is fairly simple, and one can learn to drive

a car reasonably easily. At the same time, cars are still very complex for developers and administrators (or car designers and car mechanics).

The main idea here is that even though a system may be very complex, internally, it is possible to hide that complexity and present relatively simple interfaces to developers, administrators and end-users. One way of doing this is through high-level programming and self-configuration and self-repair. The main challenge is to make high-level programming, self-configuration and self-repair reliable and predictable enough that users do not have to look under the hood too often. This is similar to the way in which car drivers do not worry about the internal functioning of the car most of the times. Of course, when things do go wrong, the systems must have some way of uncovering the different layers of abstraction and allow some form of debugging.

The complexity metrics we have proposed try to measure the complexity of using a system to perform tasks, as opposed to measuring the internal complexity of the system. Of course, "using" a system means different things for developers, administrators and end-users. For developers, it refers to using a middleware, programming language or framework to develop new applications or services. For administrators, it refers to using system management tools, command line interfaces or scripts to manage or configure a system. For end-users, it refers to using different kinds of user interfaces for interacting with applications or services while performing tasks.

Chapter 9

Related Work

The Aura Project[1] represents user tasks at a high level and then maps each task to applications and devices available at a location. It also has a notion of utility to discover the best mapping. However, it does not have mechanisms for learning user preferences or taking into account security and other policies during task execution.

The iROS[21] system is based on an Event Heap and uses soft-state maintenance and fast restart to recover from failures. It, however, does not optimize performance of tasks or discover alternate ways of performing tasks in case of failures. The Activity-Centered Computing project [2] handles activities as first class objects and allows users to suspend and resume activities. The task-computing model [3] allows a user to specify a behavior as a set of tasks that need to be completed using service descriptions. The system determines the way the tasks are to be composed. The operator graph model [4] uses a programming model where services to be composed are specified as descriptions and interactions among services are defined using operators. MIT's Oxygen Project[6] automatically satisfies abstract user goals by assembling, on-the-fly, an implementation that utilizes the resources currently available to the user However, these approaches do not have any mechanisms for choosing the best way of composing services, learning user preferences or self-repairing in the case of failure.

A related concept to task execution is workflows. Workflows define the sequence of tasks to execute to achieve some goal. They are used to automate business processes, in whole or in part, and allow passing documents, information, or tasks from one participant to another for action, according to a set of procedural rules. Languages such as BPEL [12] are used to define the set of actions (in terms of invocations of web services) that are required to achieve some goal. However, the limitation of most workflow systems is that workflow scripts are static in nature and cannot adapt dynamically to changing resource availabilities or different contexts.

In the area of web services, there has been related work done on the automatic composition of web services for achieving a goal. [28] describes the use of an augmented Golog interpreter, which is based on situation calculus, to determine a sequence of semantic web services to be executed for achieving a certain goal. The Sword toolkit [29] uses a rule-based expert engine for determining how to construct a composite web service from existing services. [30] and [31] propose planners that use STRIPS and SHOP2 respectively to compose a plan, given the goal and DAML-S descriptions of a set of basic services. However, all these approaches also do not have mechanisms for self-repair, for choosing the best way of composing services or for interacting with end-users and learning their preferences.

The QCompiler [41] is a meta-data QoS programming and compilation framework for developing and deploying quality aware applications. It, however, does not have a way of comparing different possible compilations or recovering from failures. The QCompiler can form a part of our autonomic task execution framework to figure out the best way of meeting QoS-based task parameters based on meta-data and application specifications. SpiderNet [42] is a quality-aware, autonomic service composition framework that allows streaming applications to be assembled, automatically, from distributed data sources, data sinks and intermediate stream processing functions. The main difference between SpiderNet and our framework is that SpiderNet aims to

optimize QoS metrics like network delay and loss probability, while our framework looks to optimize many dimensions of utility including usability, user satisfaction and various performance metrics. Also the space of possible actions (or services) in SpiderNet includes only operations on multimedia streams, while actions in our framework could be interactions with different types of devices, services, applications and users. Another related work in the area of QoS-aware service composition is [43], which tries to optimize the performance related aspects (e.g. network bandwidth and path delay) of composite services. In particular, it tries to reduce redundancies in data delivery and service execution through explorations of different types of multicast (service multicast and data multicast). Again, such optimizations can be plugged into our autonomic task execution framework for optimizing QoS parameters of a task that utilizes composite web services. Our framework can, in some sense, act as a plugging board for different kinds of optimization algorithms that optimize different kinds of task parameters.

In the area of Autonomic Computing, the Accord Programming Framework [7] allows the development and composition of autonomic components through workflows. It, however, does not address issues relating to optimizing or repairing workflows. The Unity system [10] uses goal-driven self-assembly to configure itself. However, the utility function [9] it uses assumes that one can quantify the utility of different choices. The ACT framework [11] allows optimizing existing CORBA applications. However, it doesn't specify generic ways of configuring and optimizing different applications.

In the area of system complexity, there has not been much work in defining or measuring the complexity of distributed systems so far. Colyer et al [50] have described the problem of middleware complexity and have shown how aspect oriented software development can simplify development. Booch [51] has also discussed why software is inherently complex and how abstractions and aspects can reduce complexity. They, however, do not propose quantitative metrics for measuring complexity.

Chapter 10

Future Work

There are a number of possible future extensions to this work. One is the development of a GUI for specifying task flowcharts. This GUI would allow developers and power users to draw the flowchart representing a task and specify the different activities in the flowchart. Such a GUI would enable rapid specification of tasks and would also allow users, who are not programmers, to develop tasks.

Our current solution is centralized in the sense that a single service orchestrates other entities to perform a user's task. We are working on a multi-agent solution where different users interact with their own agents for performing tasks. This will allow multiple users to perform tasks in the same environment while resolving conflicts if they arise.

An important requirement of the system is a good policy development and management framework that allows developers, administrators and end-users to write policies that affect system behavior and that helps them ensure that different policies are consistent and non-conflicting. Hence, good interfaces for specifying policies are required in order to enable different kinds of users to frame useful policies that can help customizing the execution of tasks to different environments and to their personal or shared preferences. Also, the interfaces need to be backed by reasoning engines that can detect when two or more policies conflict. We have already started

some preliminary work in this regard [47]. There also needs to be some sort of simulator where users can experiment with different policies and ensure that they behave according to their expectations.

The success of the framework does rely on the ability to specify how a process or a task should proceed as a flowchart or in a workflow-like manner. While there are many scenarios where it is possible to specify such graphs or workflows, there are other environments and scenarios where the user needs to have more freedom in interacting with the ubiquitous computing environment in a more spontaneous and non-deterministic manner. In such scenarios, users may not have specific goals in mind, but instead may want to explore the environment. An interesting area of research is to come up with task models that allow such spontaneous behavior, rather than constraining the user to follow the steps specified by the task developer.

In the area of complexity metrics, we would like to evaluate different kinds of distributed computing systems, apart from ubiquitous computing systems, using the complexity metrics. In particular, we would like to evaluate other large, complex distributed systems like sensor networks, enterprise systems and internet-based systems. This would help establish the generality of the metrics, both for different classes of people (developers, administrators and end-users) as well as for different kinds of distributed computing systems.

Another area of future work is to try to extend the set of metrics to include a measure for debugging and failure-recovery. Distributed computing systems may fail due to a number of reasons - OS faults, software bugs, hardware errors, human error, etc. While self-repair and high-level programming can reduce the occurrence of failures and provide automated, graceful failure-recovery, systems may still fail and require human intervention. Besides, self-configuring and self-repairing systems may misbehave or take wrong actions. Hence, distributed computing systems should provide mechanisms for uncovering the different layers of abstraction and deducing

the cause of faults. The complexity of debugging a system is an important measure of the ease of using and maintaining the system.

System complexity forms just one of the factors that influence the productivity of developers, administrators and end-users. Other factors that affect productivity include communication and interaction between different developers, administrators and end-users, ambiguous and changing requirements, technology churn, etc. While we are still far from being able to quantify these other factors, they would all form part of any evaluation of the productivity of different kinds of users.

Chapter 11

Conclusions

In this thesis, we have presented a high-level task execution framework that enables autonomic ubiquitous computing. The framework automatically or semi-automatically configures the ubiquitous computing environment in the best way for performing various tasks and also recovers from failures. Some of the key features of the framework are the use of ontologies for specifying hierarchies of entities and their properties, the use of learning to customize task execution, incorporation of security and other policies, and the use of a generic, parameterized task model that allows the same tasks to be run in different environments with different resources.

The core contributions of this thesis are a set of solutions for autonomic ubiquitous computing (a high-level programming model, a semantic discovery process and a task execution framework that allows self-configuration, self-optimization and self-repair). Besides, the other two key contributions of this thesis are an exploration of different levels of automation that can be carried out in distributed computing systems and the proposal of a set of aspects and metrics for characterizing and evaluating the complexity of distributed computing systems. These two aspects of the thesis, we believe, go beyond the confines of ubiquitous computing and have potential impact on different areas of computer systems research and development.

One of the key uses of computers, in general, is automating processes that were

originally performed by humans. Computers allow humans to perform increasingly higher and higher cognitive functions and tasks. They take care of performing repetitive, "simple" functions, leaving the human mind free to engage in "higher" pursuits. The issue of automation raises two important questions that have always stayed with and that will continue to stay with computer science research: how to automate and how much to automate. In this thesis, we have explored these two questions in the area of ubiquitous computing. These questions, however, exist in a wide variety of areas in computer science such as intrusion detection, human computer interaction, robotics, network management, policy management, data integration, natural language translation and processing, speech-to-text and text-to-speech, etc. Different areas have their own algorithms and strategies to automate. However, the question of how much to automate shares some common features - it depends on the accuracy and reliability of the algorithms in question, the criticality of the task at hand, and the amount of control the user needs to have. More research needs to be carried out to come up with methodologies or design patterns to help answer the question of how much to automate, and also possibly, to discover a grand unified theory of automation for different areas of computer science.

The issue of complexity is another one that impacts different types of computer systems. In this thesis, we have made a start in coming up with different aspects of complexity and ways of measuring these aspects. We have also identified several design principles that can, potentially, help reduce certain aspects of complexity. However, a lot of more research needs to be done to polish the different metrics, understand how the different aspects of complexity impact various kinds of systems, figure out the best way of comparing different systems using these metrics, and finally, comping up with a complexity analyzing tool that system designers and architects can make use of while designing and architecting systems.

Appendix A

Example slideshow task in C++ consisting of various activities

```
#include "ApplicationActivities.h"
voidPresentationTask() {
    ApplicationActivities appAct = new ApplicationActivities();
    ActivityQueryStructure aqs[3];
    aqs[0] = *(appAct->createActivityQueryStruct("coordinator", 1, 1, "Device"));
    aqs[1] = *(appAct->createActivityQueryStruct("model", 1, 3, "Device",
                                                  "filename", "Class"));
    aqs[2] = *(appAct->createActivityQueryStruct("application", 1, 2,
                                                  "Number of Presentations",
                                                  "Number of Controllers"));
    appAct->queryForTaskParameters(3, aqs);
    if (appAct->checkIfAppRunning("pptapp") == TRUE) {
      ActivityQueryStructure aqs1[1];
      aqs1[0] = *(appAct->createActivityQueryStruct("application", 1, 1,
          "Start New Application"));
      appAct->queryForTaskParameters(1, aqs1);
```

```
if (!strcmp(appAct->getParamValue("application","Start New Application"), "1") {
    ActivityQueryStructure aqs2[1];
    aqs2[0] = *(appAct->createActivityQueryStruct("application",1,1,"instance"));
    appAct->queryForTaskParameters(1, aqs2);
    appAct->changeFileGivenIOR(
                    appAct->getParamValue("application","instance"),
                    appAct->getParamValue("model","filename") );
    }
}
else
{
    ActivityQueryStructure aqs3[2];
    aqs1[0] = *(appAct->createActivityQueryStruct("presentation",
                    appAct->getParamValue("application","Number of Presentations"),
                    2, "Device","Class"));
    aqs1[0] = *(appAct->createActivityQueryStruct("controller",
                    appAct->getParamValue("application","Number of Controllers"),
                    2, "Device","Class"));
    appAct:queryForTaskParameters(2, aqs3);
    char* appName = appAct->generateAppName();
    appAct->createCoordinator(appName,
                    appAct->getParamValue("coordinator","Device"));
    char* modelClass = appAct->getParamValue("model","Class");
    char* modelargs = appAct->makeModelArgs(modelClass,
                    appAct->getParamValue("model","filename"));

    appAct->createModel(appName, modelClass,
```

```
                    appAct->getParamValue("model","Device"), modelargs);

        for(i=0;i<appAct->getParamValue("application","Number of Presentations")
          appAct->createPresentation(appName,
                     appAct->getParamValue("presentation","Class",i),
                     appAct->getParamValue("presentation","Device",i) );

        for(i=0;i<appAct->getParamValue("application","Number of Controllers")
          appAct->createPresentation(appName,
                     appAct->getParamValue("controller","Class",i),
                     appAct->getParamValue("controller","Device",i) );

      }
    delete(appAct);
}
```

Appendix B

Segment of Parameter XML File for a slideshow task

```
<!--Describing parameters associated with the model
  - i.e. device on which model runs and name of slideshow file -->
<Entity name="model">
  <Parameter>
    <Name>Device</Name>
    <Class>Device</Class>
    <Mode>Automatic</Mode>
    <Metric>Space Policy</Metric>
  </Parameter>
  <Parameter>
    <Name>filename</Name>
    <Class>SlideshowFile</Class>
    <Mode>Manual</Mode>
  </Parameter>
</Entity>
```

```
<!--Describing parameters associated with the overall application
  - i.e. number of presentations and number of controllers-->
<Entity name="application">
  <Parameter>
    <Name>Number of presentations</Name>
    <Class>Number</Class>
    <Mode>Manual</Mode>
  </Parameter>
  <Parameter>
    <Name>Number of controllers</Name>
    <Class>Number</Class>
    <Mode>Manual</Mode>
  </Parameter>
</Entity>

<!--Describing parameters associated with the presentation
  - i.e. device on which the presentation will runs and
  the class of the presentation component
  (e.g. PowerPoint or AcrobatReader-->
<Entity name="presentation">
  <Parameter>
    <Name>Device</Name>
    <Class>Visual Output</Class>
    <Property>
      <PropName>resolution</PropName>
      <PropValue>1600*1200</PropValue>
    </Property>
```

```
    <Mode>Manual</Mode>
  </Parameter>
  <Parameter>
    <Name>Class</Name>
    <Class>SlideShowPresentation</Class>
    <Mode>Automatic</Mode>
    <Metric>Space Policy</Metric>
  </Parameter>
</Entity>
```

Appendix C

Defining the Learning Feature Space

The following of the learning metadata XML file defines the different features to be used in the learning process.

```
<FeatureSet>
<LearningFeature>
  <Name>Coordinator Device</Name>
  <Type>Device</Type>
  <Start>1</Start>
</LearningFeature>

<LearningFeature>
  <Name>Model Device</Name>
  <Type>Device</Type>
  <Start>41</Start>
</LearningFeature>

<LearningFeature>
```

```
    <Name>Model Filename</Name>
    <Type>Filename</Type>
    <Start>81</Start>
  </LearningFeature>

  <LearningFeature>
    <Name>Number of Presentations</Name>
    <Type>Number</Type>
    <Start>281</Start>
  </LearningFeature>

  <LearningFeature>
    <Name>Number of Controllers</Name>
    <Type>Number</Type>
    <Start>291</Start>
  </LearningFeature>

  <LearningFeature>
    <Name>Presentation Device</Name>
    <Type>Device</Type>
    <Start>301</Start>
  </LearningFeature>

  <LearningFeature>
    <Name>Presentation Class</Name>
    <Type>PresentationClass</Type>
    <Start>341</Start>
```

```
</LearningFeature>

<LearningFeature>
  <Name>Controller Device</Name>
  <Type>Device</Type>
  <Start>361</Start>
</LearningFeature>

<LearningFeature>
  <Name>Controller Class</Name>
  <Type>ControllerClass</Type>
  <Start>401</Start>
</LearningFeature>

<LearningFeature>
  <Name>Name of User</Name>
  <Type>User</Type>
  <Start>501</Start>
</LearningFeature>

<LearningFeature>
  <Name>People Present in Room</Name>
  <Type>User</Type>
  <Start>541</Start>
</LearningFeature>

<LearningFeature>
```

```
  <Name>Devices Running</Name>
  <Type>Device</Type>
  <Start>581</Start>
</LearningFeature>

<LearningFeature>
  <Name>Applications Running</Name>
  <Type>Application</Type>
  <Start>621</Start>
</LearningFeature>

<LearningFeature>
  <Name>Time of Day</Name>
  <Type>Time</Type>
  <Start>661</Start>
</LearningFeature>

<LearningFeature>
  <Name>Activity in Space</Name>
  <Type>Activity</Type>
  <Start>673</Start>
</LearningFeature>

<LearningFeature>
  <Name>Kind of Task</Name>
  <Type>Task</Type>
  <Start>681</Start>
```

```
</LearningFeature>

<LearningFeature>
  <Name>Application Running on Device</Name>
  <Type>Application</Type>
  <Start>721</Start>
</LearningFeature>
</FeatureSet>
```

Appendix D

Defining Labels of Learning Feature Values

The following segment of the learning metadata XML file defines labels for different values.


```
<Map>
<Instance type="Device">
<Device>
<Name>srg-a21p-3-laptop</Name>
<ID>0</ID>
</Device>
<Device>
<Name>cs-pc2401-1</Name>
<ID>1</ID>
</Device>
<Device>
<Name>cs-pc2401-2</Name>
<ID>2</ID>
</Device>
```

```
<Device>
<Name>AgentSmithius-laptop</Name>
<ID>14</ID>
</Device>
</Instance>

<Instance type="PresentationClass">
<PresentationClass>
<Name>CORBA/PPTPresentation</Name>
<ID>0</ID>
</PresentationClass>
<PresentationClass>
<Name>CORBA/MP3Player</Name>
<ID>1</ID>
</PresentationClass>
</Instance>
<Instance type="User">
<User>
<Name>anand</Name>
<ID>0</ID>
</User>
<User>
<Name>jalal</Name>
<ID>1</ID>
</User>
</Instance>
```

```
<Instance type="Task">
<Task>
<Name>unknown</Name>
<ID>0</ID>
</Task>

<Task>
<Name>PresentationTask</Name>
<ID>1</ID>
</Task>
<Task>
<Name>MusicTask</Name>
<ID>2</ID>
</Task>
</Instance>
</Map>
```

Appendix E

Learning Training File

The following is a segment of an example training file generated by the Logger.

```
  1,41,282,292,301,341,361,401,501,581,621,621,669,673,682:
1,41,282,292,301,341,361,401,501,581,671,673,682:
1,41,282,292,301,341,361,401,501,581,620,671,673,682:
1,41,282,292,301,341,361,401,501,581,620,621,671,673,682:
1,41,282,292,301,341,361,401,501,581,620,671,673,682:
1,41,282,292,301,341,361,401,501,581,620,668,673,682:
1,41,282,292,301,341,361,401,501,581,668,673,682:
1,41,282,292,301,341,361,401,501,581,621,668,673,682:
1,41,282,292,301,341,361,401,501,581,621,620,621,668,673,682:
1,41,282,292,301,341,361,401,501,581,668,673,682:
1,41,282,292,301,341,361,401,500,581,620,669,673,682:
1,41,283,292,301,341,301,341,361,401,500,581,620,621,669,673,682:
1,41,282,293,301,341,361,401,361,401,500,581,620, 621,669,673,682:
1,41,282,292,301,341,361,401,500,581,621,620,669,673,682:
1,41,282,292,301,341,361,401,501,581,620,621,668,673,682:
1,41,282,292,301,342,361,402,500,581,620,622,669,673,683:
1,41,282,292,301,342,361,402,500,581,620,622,669,673,683:
```

```
1,41,282,292,301,342,361,402,500,581,620,622,669,673,683:
1,41,282,292,301,342,361,402,500,581,620,622,669,673,683:
1,41,282,292,301,342,361,402,500,581,622,620,669,673,683:
1,41,282,292,301,341,361,401,500,581,620,669,673,682:
```

References

[1] J. P. Sousa, D. Garlan, “Beyond Desktop Management: Scaling Task Management in Space and Time” Technical Report, CMU-CS-04-160, School of Computer Science, Carnegie Mellon University

[2] H.B. Christensen, J.E. Bardram, “Supporting Human Activities - Exploring Activity-Centered Computing”. In Proceeding of Ubiquitous Computing 2002

[3] Z. Song, et al, “Dynamic Service Discovery and Management in Task Computing”, in MobiQuitous’04, 2004.

[4] G. Chen, M. Li, and D. Kotz, “Design and Implementation of a Large-Scale Context Fusion Network”, presented at First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous’04), 2004.

[5] M. Roman, at al, “Application Mobility in Active Spaces”, In 1st International Conference on Mobile and Ubiquitous Multimedia, Oulu, Finland, 2002.

[6] U. Saif, et al “A Case for Goal-oriented Programming Semantics”. In System Support for Ubiquitous Computing Workshop at Ubicomp 2003, Seattle, WA, Oct 12, 2003

[7] H. Liu et al “A Component-Based Programming Model for Autonomic Applications”. in ICAC 2004, New York, NY

[8] T.J. McCabe and C.W. Butler,. “Design Complexity Measurement and Testing.” Communica-tions of the ACM 32, 12 (December 1989): 1415-1425

[9] W.E.Walsh et al “Utility Functions in Autonomic Systems” in ICAC 2004, New York, NY

[10] D. Chess et al “Unity: Experiences with a Prototype Autonomic Computing System” in ICAC 2004, New York, NY

[11] S.M. Sadjadi et al “Transparent Self-Optimization in Existing CORBA Applications” in ICAC2004, New York, NY

[12] “Business Process Execution Language for Web Services Version 1.0”, BEA, IBM and Microsoft, August 2002: http://www-106.ibm.com/developerworks/library/ws-bpel/

[13] M. Roman, et al, “Gaia: A Middleware Infrastructure to Enable Active Spaces”, IEEE Pervasive Computing Magazine, vol. 1, pp. 74-83, 2002.

[14] W. Asprey, et al., “Conquer System Complexity: Build Systems with Billions of Parts”, in CRA Conference on Grand Research Challenges in Computer Science and Engineering, Warren-ton, VA (2002), pp. 29-33,

[15] N. F. Noy, et al, “Creating Semantic Web Contents with Protege-2000”, IEEE Intelligent Systems, vol. 16, pp. 60-71, 2001.

[16] A. Ranganathan, et al, “Olympus: A High-Level Programming Model for Pervasive Comput-ing Environments”, in IEEE PerCom 2005, Kauai Island, Hawaii, 2005

[17] M. Halstead. Elements of Software Science, Operating, and Programming Systems Series Volume 7, Elsevier, 1977

[18] A. J. Carlson, et al, “SNoW User’s Guide.” UIUC Tech report UIUC-DCS-R-99-210, 1999

[19] R. Ierusalimschy et al, “Lua: An Extensible Extension Language”, Software: Practice and Experience Journal., Vol 26, No. 6, 1996, pp 635-652.

[20] M. Dean, et al, “OWL web ontology language 1.0 reference”, http://www.w3.org/TR/owl-ref/, 2002.

[21] S.R. Ponnekanti, et al. “Portability, Extensibility and Robustness in iROS”, PerCom 2003

[22] B. McBride, “Jena: A Semantic Web Toolkit”, IEEE Internet Computing archive, vol. 6, pp. 55 - 59, 2002.

[23] “XSB Prolog.” http://xsb.sourceforge.net

[24] A.K. Dey, et al. “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications”, anchor article of a special issue on Context-Aware Computing, Human-Computer Interaction (HCI) Journal, Vol. 16, 2001.

[25] T. Berners-Lee et al. “A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities” http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html

[26] N. Guarino “Formal Ontology in Information Systems” Proc. of FOIS’98, Trento, Italy

[27] C.K.Hess, M. Roman, R.H. Campbell, Building Applications for Ubiquitous Computing Environments In International Conference on Pervasive Computing (Pervasive 2002), pp. 16-29, Zurich, Switzerland, August 26-28, 2002.

[28] S. McIlraith and T. Son. "Adapting Golog for Composition of Semantic Web Services". In Proceedings of the Eighth International Conference on Knowledge Representation and Reason-ing (KR2002), Toulouse, France, 2002.

[29] S. R. Ponnekanti and A. Fox. "SWORD: A Developer Toolkit for Web Service Composition". In Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI, 2002.

[30] M. Sheshagiri, et al. "A Planner for Composing Services Described in DAML-S". In Work-shop on Planning for Web Services, International Conference on Automated Planning and Scheduling, Trento, July 2003

[31] J. Hendler et al. "Automating DAML-S Web Services Composition using SHOP2". In 2nd In-ternational Semantic Web Conference (ISWC2003) Florida, October 2003

[32] A. Ranganathan, et al, "MiddleWhere: A Middleware for Location Awareness in Ubiqui-tous Computing Applications", In ACM/IFIP/USENIX 5'th International Middleware Confer-ence, Toronto, Canada, Oct 18-22, 2004

[33] L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology", presented at WWW 2003, 2003.

[34] J. Gonzalez-Castillo and e. al., "Description Logics for Matchmaking Services", HP Labo-ratories Bristol, Bristol HPL-2001-265 2002.

[35] G. E. Krasner and S. T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System", Journal of Object Oriented Programming, vol. 1, pp. 26-49, 1988.

[36] A. Ranganathan and R. H. Campbell, "A Middleware for Context-Aware Agents in Ubiqui-tous Computing Environments", In ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, Jun 16-20, 2003

[37] OMG, "CORBA, Architecture and Specification", Common Object Request Broker Archi-tecture (CORBA) 1998.

[38] A. Ranganathan, et al, "Ontologies in a Pervasive Computing Environment", In Workshop on Ontologies and Distributed Systems (part of IJCAI 2003), Acapulco, Mexico, Aug 9 2003

[39] J. Kephart and D. Chess, "The Vision of Autonomic Computing", IEEE Computer, Vol. 36, No. 1 (2003), pp. 41-50

[40] D. McDermott, and the AIPS-98 Planning Competition Committee. "PDDL - The Planning Domain Definition Language", Draft 1.6, June 1998

[41] D. Wichadakul, "Q-Compiler: Meta-Data QoS-Aware Programming and Compilation Framework", Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, January, 2003

[42] Xiaohui Gu, "SpiderNet: A Quality-Aware Service Composition Middleware", Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October, 2004.

[43] J. Jin, K. Nahrstedt, "On Exploring Performance Optimizations in Web Service Composition", in Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware 2004), Toronto, Canada, October, 2004

[44] W. Celes, "toLua - accessing C/C++ code from Lua", http://www.tecgraf.puc-rio.br/ celes/tolua/

[45] J. Al-Muhtadi, S. Chetan, A. Ranganathan and R. Campbell, "Super Spaces: A Middleware for Large-Scale Pervasive Computing Environments", Perware '04: IEEE International Workshop on Pervasive Computing and Communications, Orlando, Florida, March 2004.

[46] M. Li and P.M.B. Vitanyi, "An Introduction to Kolmogorov Complexity and its Applications", Springer-Verlag, New York, 2nd Edition, 1997

[47] C. S. Shankar, A. Ranganathan and R. Campbell, "An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environment", In Mobiquitous 2005: The Second Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services, San Diego, California, July 2005.

[48] A. Ranganathan, R. E. McGrath, R. H. Campbell, M. D. Mickunas, "Use of Ontologies in a Pervasive Computing Environment" , In The Knowledge Engineering Review, Vol 18:3, 209-220, 2004, Cambridge University Press.

[49] A. G. Cohn, B. Bennett, J. M. Gooday, and N. Gotts, "RCC: a calculus for Region based Qualitative Spatial Reasoning," In GeoInformatica, 1 , pp 275-316, (1997).

[50] A. Colyer, G. Blair and A. Rashid, "Managing Complexity in Middleware", In The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), In AOSD 2003, Boston, MA, Mar 17, 2003

[51] G. Booch, "The Complexity of Programming Models". Keynote talk at AOSD 2005, Chicago, Il, March 14-18, 2005

[52] D.E. Kieras, "Towards a practical GOMS model methodology for user interface design". In M. Helander (Ed.), Handbook of Human-Computer Interaction (pp. 135-158), 1988. Amsterdam: North-Holland Elsevier.

[53] S. Henry and D. Kafura, "Software structure metrics based on information flow". In IEEE Transactions on Software Engineering, 7(5):510–518, Sept. 1981

[54] M. Page-Jones, "The Practical Guide to Structured Systems Design". Yourdon Press, New York, NY, 1980.

[55] G. Myers, "Reliable Software Through Composite Design". Mason and Lipscomb Publishers, New York, NY, 1974.

Author's Biography

Anand Ranganathan was born in Bangalore, India on Dec 10, 1978. He received his B.Tech. in Computer Science and Engineering in 2000 from the Indian Institute of Technology in Madras. He then moved to Champaign, IL to pursue his graduate studies. He finished his PhD in 2005 at the Department of Computer Science in the University of Illinois at Urbana-Champaign, under the supervision of Prof. Roy H. Campbell. His research interests include ubiquitous computing, autonomic computing, middleware, the Semantic Web, distributed systems, artificial intelligence and social networks. He has been involved in the development of Gaia, an infrastructure for programming and managing ubiquitous computing environments. He received the C.W. Gear Outstanding Graduate Student Award from the Department of Computer Science, University of Illinois at Urbana-Champaign, in 2004.