Olympus: A High-Level Programming Model for Pervasive Computing Environments[1]

Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, M. Dennis Mickunas
University of Illinois at Urbana-Champaign
{ranganat, chetan, almuhtad, rhc, mickunas}@uiuc.edu

Abstract

Pervasive Computing advocates the enhancement of physical spaces with computing and communication resources that help users perform various kinds of tasks. We call these enhanced physical spaces Active Spaces. Active Spaces are highly dynamic – the context and resources available in these environments can change rapidly. The large number of entities present in these spaces and the dynamism associated with them make it difficult for developers to program these environments. It is not always clear at development time which resources are to be used for performing various kinds of tasks and how to use them. In this paper, we introduce a new high-level programming model for pervasive computing environments, Olympus. The main feature of this model is that developers can specify Active Space entities and common Active Space operations at an abstract, high level. Active Space entities (which include services, applications, devices, physical objects, locations and users) can be specified using high level descriptions. Our framework resolves these descriptions into actual Active Space entities based on constraints specified by the developer, ontological descriptions of entities, the resources available in the current space, space-level policies and the current context of the space. The programming model also provides the developers with operators for commonly used functions. Examples of operators include start, stop and move components. Thus, developers do not have to worry about how various tasks are performed in the space in which their program is to be deployed. These details are taken care of by the model and the developer is free to focus on the actual logic of the program. In this paper, we discuss the programming model, its implementation and several example Active Space programs that have been developed using this model.

1. Introduction

Pervasive Computing envisions a world with users interacting naturally with device-rich, context-aware environments to perform various kinds of tasks. However, these environments are highly dynamic. The resources (which include devices, services and applications) available in these environments, as well as the context of these environments, can change rapidly. Thus, programs running in these environments must be able to adapt to the changing contexts and resource availabilities. This places a large burden on the developer in specifying how the program should behave in different contexts and when different kinds of resources are available. Besides, different environments may have different ways of performing the same kind of task. The developer cannot be expected to know how various tasks are to be performed in different environments. This places a bottleneck on the rapid development and prototyping of new services and applications in these environments.

Our notion of a pervasive computing environment is a physical space that has been enhanced with a large number of digital devices such as various sensors, computers and actuators. We call this space an Active Space. In previous research, we developed a middleware, Gaia [1], for programming Active Spaces. We have deployed Active Spaces in a number of rooms in our Computer Science building. One of the problems we came across was that many of our programs were not portable across different spaces. Developers, often, had to customize their applications and services for new spaces because different spaces had different kinds of resources. Besides, different spaces had different policies regarding the usage of resources for performing various kinds of tasks. Developers had to be aware of these policies while developing their services and applications. Finally, because Active Spaces are characterized by a large number of different types of services and applications, there are often different ways of performing the same task. However, some ways are better than others depending on the current context, resources available and user preferences. Hence, programs in Active Spaces 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. Thus, we came to the conclusion that it is necessary to provide developers with a higher level of abstraction while programming these

[1] This research is supported by a grant from the National Science Foundation, NSF CCR 0086094 ITR and NSF 99-72884 EQ.

spaces, so that they need not be aware of the specific resources available, context, policies and user preferences while developing their programs. In particular, we need high-level abstractions for referring to Active Space entities and operations.

On the basis of these requirements, we have developed a new high-level programming model called *Olympus*. Olympus allows developers to specify Active Space entities at an abstract, high level. Active Space entities include services, applications, devices, physical objects, locations and users. Developers using the Olympus programming model refer to these entities as virtual entities in their programs. The Olympus model is associated with a framework, which takes care of resolving these virtual entities into actual Active Space entities based on constraints specified by the developer, the resources available in the current space, space-level policies and the current context of the space. The framework makes use of ontological hierarchies and descriptions of entities to allow semantic discovery of appropriate entities. The ontological hierarchies of entities also aids the development process since developers can now browse the ontologies to see what kinds of entities are available and what kinds of constraints they can specify on these entities. Finally, the framework makes use of a utility model in order to choose the "best" entity available in a space for performing a certain kind of task, in case there are many choices.

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 Olympus framework first discovers possible classes of entities that satisfy all the requirements. Then, it discovers instances of these classes that satisfy instance-level requirements. Finally, it ranks the instances based on a utility function and chooses the "best" one(s). 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.

The other main feature of Olympus is that common Active Space operations can also be specified at a high-level. Common Active Space operations include starting, stopping and moving components; notifying users; taking actions when users enter or exit a space, etc. These operations are modeled as operators in the Olympus programming model. Developers do not have to worry about how these operations are performed in the space in which their program is deployed. The Olympus framework determines how these operators are executed using various services present in the space.

We have developed the Olympus programming model as part of our Gaia middleware. Currently, the model is implemented in C++. Developers can specify virtual entities and Active Space operators in their C++ programs and execute them in different spaces. Table 1 shows a sample program developed using the model. In the program, the developer just says that he wants to start a slideshow application with the file Olympus.ppt in the space where the user Sal is located. The framework takes care of choosing exactly how to start the application. It chooses which components to use: for example, MS PowerPoint could be used, or Acrobat Reader, in case there are no Windows machines available. It takes care of converting data formats (from ppt to pdf). It chooses the "best" devices to display the slideshow in case many are available. It also decides the "best' way of controlling the slideshow (e.g. using a GUI or by using voice commands). These decisions are made based on developer-specified constraints, policies and user preferences.

Table 1. Sample Olympus program

```
ActiveSpace as1;     //refers to a virtual active space entity

as1.hasProp("containsPerson","Sal");
as1.instantiate();     /*as1 now refers to the Active Space
                         where the user Sal is located */

Application app1;    //refers to a virtual application entity
app1.hasProp("class","Slideshow");
app1.hasProp("file","Olympus.ppt");
app1.start(as1);      /*app1 is started in Active Space as1
                        in the "best" possible configuration */
```

The main contribution of the paper is the proposal of a high-level programming model that allows developers to program Active Spaces without worrying about how common operations are implemented in the space and which are the best resources for executing a task. This enables rapid prototyping and testing of different applications. Some of the key features of this model are:

1. Ontologies for specifying hierarchies of different kinds of entities and for specifying entity properties. Developers can browse the ontologies to discover classes of entities that can be used in his programs.
2. A novel semantic matching algorithm based on ontological hierarchies for discovering appropriate classes of entities for performing a certain task.
3. Context-aware policies and rules for choosing appropriate classes and instances of entities.
4. A multi-dimensional utility function for choosing the "best" instances of entities for performing a certain task.
5. Proposal of a basic set of high-level operators for programming pervasive computing environments.

In the rest of the paper, we describe the details of Olympus. We first introduce Active Spaces and our middleware, Gaia. Section 3 gives an overview of the elements of the Olympus Programming Model. Section 4 goes into the details of the discovery process. Section 5 describes the high-level operators in our model. Section 6 has some example programs. Next we describe the implementation details and our experiences in using this

model. Finally, we conclude with a discussion of related work and future work.

2. Active Spaces and Gaia

An *Active Space* is a physically-bounded collection, such as a room, of devices, objects, users, services and applications. A meta-operating system, called *Gaia* [1], manages resources of an 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 also has a context infrastructure [2] that allows obtaining the current context of the space. Gaia uses CORBA [3] to enable distributed computing.

3. The Olympus Programming Model

A programming model is defined as an abstract machine, providing certain data types and operations to the programming level above, and requiring implementations for each of these operations on all of the architectures below [4]. Active spaces differ in the resources they provide, the services they support and even in the implementations of these services. Therefore, each active space can be viewed as a different architecture.

3.1 Virtual Entities

Olympus allows a user to program an Active Space in terms of virtual entities. Virtual entities are essentially variables that have not yet been instantiated. They are associated with a class defined in an ontology. Developers can specify certain properties that must be satisfied by the concrete entity, and the Olympus framework takes care of discovering appropriate entities that satisfy developer constraints as well as other constraints in ontologies and user and space policies.

There are eight basic types of entities in our model: *Application, ApplicationComponent*, *Device, Service*, *Person*, *PhysicalObject*, *Location* and *ActiveSpace*. Entities are treated as first-class objects in the model – that is they can be stored in variables, used in expressions and passed as parameters to functions.

For example, the following piece of code represents a virtual application component entity that can display a slideshow on a Linux machine.

```
ApplicationComponent app1;     //app1 is the virtual entity
app1.hasProp(“class”,”SlideShow”);
app1.hasProp(“requiresOS”,”Linux”);
```

Depending on the components available in the space, the policies of the space and user preferences, the virtual entity app1 may get instantiated with either Acrobat Reader or Ghostscript. The developer though, does not have to worry about the actual instance the entity gets instantiated with.

3.2 High-level Operators

The Olympus framework implements some commonly used Active Spaces operations. These operations are specified as operators in Olympus. Developers can use these operators in their high-level programs. The Olympus framework takes care of performing the operations by invoking appropriate services present in the space. The operator set of an Active Space is analogous to the instruction set of a computer, where we consider an Active Space to be similar to a distributed computer.

As an example, different spaces may have different ways of checking whether a user is in the space or not. Some spaces require contacting the Authentication Service to see if the user has authenticated himself using some mechanism to the space. Other spaces may have a Location Services that detect the user’s presence using a device he carries. The Olympus model hides these details from the developer and allows him to check the user’s presence using the operator ‘*in’*. The operators are executed by using libraries implementing the operators. Different libraries are used in different spaces.

4. The Discovery Process

An important element of the programming model is choosing appropriate values for the virtual entities. There are various types of constraints that need to be satisfied while choosing appropriate instances of the virtual entities. These are:

1. Constraints on the value of the variable specified by the application developer in the high-level program
2. Constraints listed in ontologies
3. Policies specified by a Space Administrator for the current space
4. User policies for the user using the program

Fig 1 provides an overview of the steps involved in discovering appropriate entities. The discovery process involves the following steps:

1. *Discovering suitable classes of entities:* The framework queries an Ontology Server in the space for classes of entities that are semantically similar to the virtual entity class. The Ontology Server returns an ordered list of classes that are semantically similar to the variable class. It uses ontologies specified in OWL [5] for determining semantic similarity.

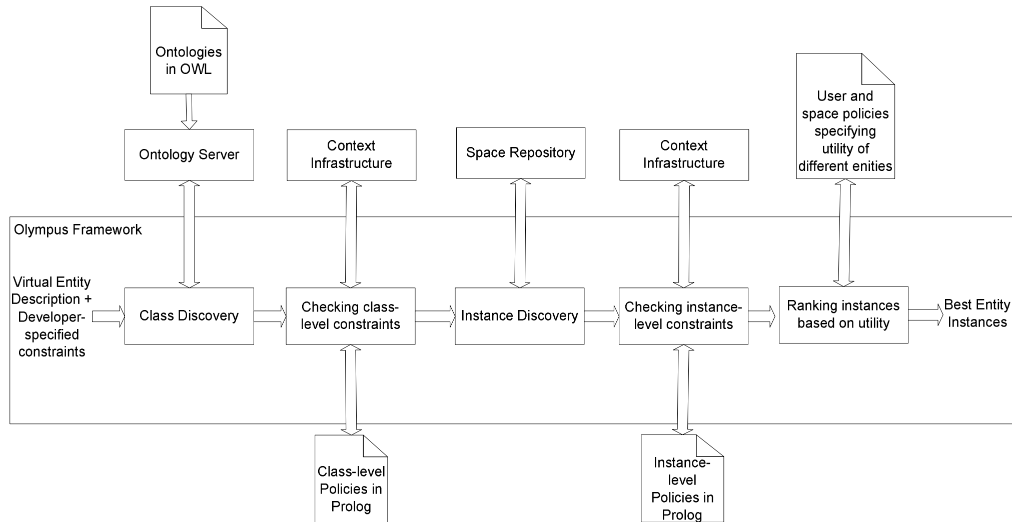

Figure 1. Discovery Process

2. *Checking class-level constraints on the similar classes:* The framework filters the list of classes returned by the Ontology Server depending on whether they satisfy class-level constraints. These class-level constraints may be specified in ontologies, policies or by the developer and may be context-sensitive. A Prolog reasoning engine is used to check the satisfaction of constraints. The Gaia Context Infrastructure is used to obtain the current context of the space.

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 available in the space. The Space Repository is a database containing information about all instances of devices, application components, services and users in the Active Space.

4. *Checking instance-level constraints on the similar classes:* For each instance returned, the framework checks if it satisfies instance-level constraints specified in the program or in the policies. The final list of instances represents possible values that the virtual entity can take.

5. *Choosing the best among the possible instances:* If only one or a few of all possible instance values are to be assigned to the virtual entity variable, then the framework chooses the most appropriate instance(s) based on a utility function.

In the following subsections, we shall describe the different parts of the discovery process in greater detail.

4.1 Ontological hierarchies of entities

Each type of entity is associated with a hierarchy defined in an ontology. We briefly describe the ApplicationComponent and Device hierarchies in order to illustrate the different kinds of hierarchies.

4.1.1 ApplicationComponent hierarchy. Applications in Gaia are built using an extension of the Model-View-Controller framework [6, 7]. Applications are made up of five components: model, presentation (generalization of view), adapter, controller and coordinator. The framework is shown in Figure 2. 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 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.

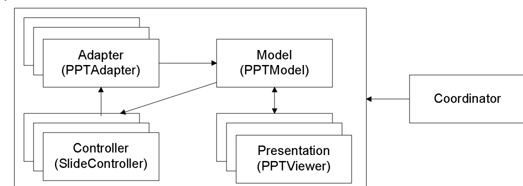

Figure 2. Application Framework and slide-show application components (in parentheses)

Fig 3 shows a portion of the hierarchy under ApplicationComponent describing different kinds of Presentation. The hierarchy, for instance, specifies two subclasses of "Presentation" – "Visual Presentation" and "Audio Presentation". It also further classifies "Visual Presentation" as "Web Browser", "Image Viewer", "SlideShow" and "Video". Ontologies allow a class to

have multiple parents –so "Video" is a subclass of both "Visual Presentation" and "Audio Presentation". This hierarchy helps in identifying how similar any two entities are. The semantic similarity of two entities can be defined in terms of where they are placed relative to one another in the hierarchy. Besides, this hierarchy offers all the advantages of a class hierarchy in an object-oriented language – for example, children entities automatically inherit the properties and constraints of the parent entities.

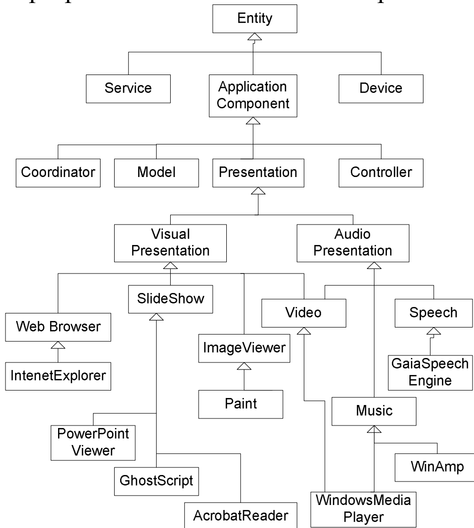

Figure 3. Presentation hierarchy in Gaia

4.1.2 Device Hierarchy. Similar to the ApplicationComponent hierarchy, we have defined a hierarchy of the different kinds of devices available in our Active Space. Fig 4 shows a portion of this hierarchy.

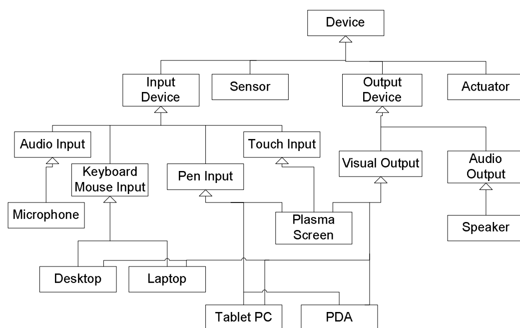

Figure 4. Device hierarchy in Gaia

4.1.2 Other Entity Type Hierarchies. Other entity types are also associated with hierarchies. The *Person* type is associated with a hierarchy representing roles of users. The *Location* type is associated with a spatial hierarchy consisting of buildings, floors, rooms, etc. The *ActiveSpace* hierarchy builds a hierarchy of spaces. The *Service* hierarchy classifies services based on the tasks they perform. Finally, the *PhysicalObject* hierarchy classified physical objects based on their functionality.

4.2 Ontological Description of entities

Each class of entity in an Active Space is associated with a description in OWL, one of the standard formats of the Semantic Web. The OWL description specifies the properties of the entity class. For example, all application component classes have an OWL description that describes various semantic properties of the component such as the tasks it can perform, the classes of devices that can host it and the data-formats it can understand.

The ontology defines relations between different concepts. One of the relations is the *requiresDevice* relationship which maps application components to a Boolean expression on devices. For example,

requiresDevice(PowerPointViewer) = PlasmaScreen ∨ Desktop ∨ Laptop ∨ Tablet PC

This means that the *PowerPointViewer* can only run on a *PlasmaScreen*, *Tablet PC* or a *Desktop*. Similarly, the *GaiaSpeechRecognizer* class in the controller hierarchy maps to the *Microphone* class in the device hierarchy.

Another relation, *requiresOS*, maps application components to operating systems. E.g.

requiresOS(PowerPointViewer) = Windows

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

4.3 Developer-specified constraints

Developers can specify constraints that the classes and instances of the virtual entities in their program must satisfy. The constraints take the form of triples (i.e. <entity> <property> <value>). Depending on the kind of property, the constraints may be on the classes of entities or on the instances. Developers can browse the ontologies to discover which properties can be specified for different kinds of entities.

For example, if the developer wishes to migrate a presentation component to the device closest to the user as he moves around, he could specify constraints on the device to be chosen in the following way:

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

The above constraints specify that the device chosen to migrate the presentation component must be of class VisualOutput (or be one of its subclasses), must be

located in the same room as the user and must support a display resolution of 800*600 pixels. If there are many candidate devices available, then the metric to use in order to choose the "best" one is based on location. In this case, the devices are ranked in ascending order of distance between the device and the user (lower ranked devices have higher preference).

4.4 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 entity classes. This notion is based on the principle that one entity 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 [8, 9]. In our algorithm, for any two concepts *C1* and *C2, C1* matches *C2* with a certain similarity-level if:

- *C1* is equivalent to *C2*, with similarity-level 0
- *C1* is a sub-concept of *C2*, with similarity-level 1
- *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, with similarity-level 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) 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 Active Space programs to adapt to the resources available in the space, even if they are vastly different from what the developer had in mind.

4.5 Space-level Policies

Space-level policies are written by the administrator of the space. These policies are written in the form of rules in Prolog. The Prolog rules specify constraints on the classes and instances of entities allowed for performing certain kinds of tasks. These rules are context-sensitive as well.

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).
```

Similarly, policies can specify instance-level constraints as well. For example,

```
disallow(Device, display, slideshow) :-
        runningVisualComponent(Display).
```

The above rule says that a certain Device instance should not be used to display slides if it is already being used to display some other visual component. Other rules and ontologies define which components are visual components. The Prolog reasoner is initialized with the state and context of the environment (such as the machines in it and their characteristics and what components they are currently hosting). The rules can also be access control rules that specify which users (specified in terms of roles) have access to a certain resource in a certain context.

4.6 Utility Function

The Olympus framework employs a multi-dimensional utility function in order to choose the best entity instance to use for performing a certain kind of task. There are four basic dimensions of utility:

1. *Location of the entity* (eg. nearer devices may be preferred to farther ones),
2. *Tasks supported by the entity*. This is based on the notion of semantic similarity of the entity to the class specified by the developer.
3. *State of the entity*. For example, devices or services with lower loads may be preferred over highly loaded devices and services.
4. *Context of the space*. For example, depending on the people in the room and the current activity, a user may prefer a different device or application for receiving notifications.

Some of the dimensions (like distance based on location) are quantitative, and allow comparing different entities to see which is better. Other dimensions (like context-based preferences) are qualitative and require the developer, the user or the space administrator to specify, in their policies, which entities are better than others in different contexts. These policies specify a partial order of the utility of entities along a dimension and thus allow comparing entities on that dimension.

Entities can have different utilities across different dimensions. A particular entity may be better than others in one dimension, but may be worse in other dimensions. Our view is that it is not possible 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. The primary dimension may be specified by the developer in the program or it may be specified in the space-level polices or in the user preferences. Just as in the case of constraints, the developer can specify how utility is to be defined for the choice of variable values. Also, space and user policies can specify the metric for utility for different kinds of variables in different contexts.

5. High-level Operators

We have identified a number of high-level operators for Active Spaces. Operators act on a target entity and can have zero or more arguments. Tables 2 and 3 list some operators with 0 and 1 arguments. Table 4 lists some event-based operators that generate events when the associated condition becomes true.

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 represented in the ontologies. The operators that have been listed in this paper are only the operators of the top-level types of entity (like service, application, device, etc.). Different subclasses of these entity types can define more specific operators. For example, Slideshow applications can define operators for navigating slides.

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

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 standard for pervasive computing environments. Such a standard would allow the same program to run unchanged in different environments.

Table 2. Operators with zero arguments for Active Spaces

Operator	Target Entity (Entity Type)	Behavior
stop	Service/ Application /ApplicationComponent	Stop service or application
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
name	Service/Application/ ApplicationComponent/ Device/PhysicalObject/Active Space	Entity name

Table 3. Operators with one argument for Active Spaces

Operator	Target Entity	Argument	Behavior
start	Application / ApplicationComponent	Device/Active Space	Starts component on appropriate device(s) or in space
resume	Application / ApplicationComponent	Device/Active Space	Resumes suspended component on appropriate device(s) or in space
in	Service/Application / ApplicationComponent/ Device/Person/ PhysicalObject/Location /Active Space	Device/Location/ Active Space	Determine containment of target entity in argument entity
migrate	Service/Application / ApplicationComponent	Device/Active Space	Migrate target entity to argument entity
deploy_in	Service/Application / ApplicationComponent	Active Space	Deploy target in a space
notify	Character string	Person	Send message to person
distance	Device/Person/PhysicalObject/ Location/Active Space	Device/Person/ Object/Location/ Active Space	Return distance between target and argument
Belongsto	Application/Service/Device/ Object	Person	Check if target is owned by argument

Table 4. Event operators for Active Spaces

Operator	Target Entity	Argument	Behavior
Enter	Person	Location/Active Space	Generate an event when Person enters argument
Exit	Person	Location/Active Space	Generate an event when Person exits argument

6. Example Programs

Programs developed on the Olympus model consist of two main segments. In the first segment, virtual entities are declared. The developer specifies the type of the entity and its properties or constraints. He, then, calls an instantiate method on the entity to ask the framework to discover appropriate instances. Developers have the choice of either getting all instances of the entity instances that satisfy the constraints or the "best" one according to a utility function specified by the developer or by policies. If the entity is declared as a list, then the framework instantiates the virtual entity with a list of all satisfying instances, else it chooses the best instance for instantiating the entity.

In the second segment, developers can use high-level operators on the entities. In case the framework returns a list of satisfying instances, the developer can iterate through the list and perform operations on instances. Tables 5 and 6 present some sample Active Space programs developed using the Olympus programming model.

Table 5. Sample Program to turn off all lights in a floor

```
DeviceList light;              /*The declaration is for a List of Devices, which means that all devices that
                                 satisfy subsequently declared properties will be part of the list */

/*Developer-specified property constraints*/
light.hasProp("class","Light");  /*light should be of the "Light" class in the ontology or a subclass of it*/
light.hasProp("location", "SiebelCenter/3");  /*light should be located in the 3'rd floor of the
                                                   Siebel Center building*/

light.instantiate();           /*this statement prompts the Olympus framework to discover all appropriate
                                  devices that satisfy the above constraints as well as other policies*/

iterator lightIter = light.iterator();
while (lightiter.hasNext()) {
        Device light1;
        light1 = lightIter.next();
        light1.on();
}
```

Table 6. Start all of Bob's suspended applications in the current active space where he is located

```
Person user1;              /*Virtual person entity*/
ActiveSpace as1;           /*Virtual active space entity*/

user1.hasProp("name","Bob");
as1.hasProp("containsPerson",user1);
user1.instantiate();       /*user1 is instantiated with a user whose name is Bob*/
as1.instantiate();         /*as1 is instantiated with the active space that contains Bob*/

ApplicationList app;       /*List of virtual application entities*/
app.hasProperty("owner",user1);
app.hasProperty("state","suspended");
app.instantiate();         /*app is instantiated with a list of applications whose owner is user1 (Bob) and
                             which are in a suspended state*/

iterator appIter = app.iterator();
while (appiter.hasNext()) {
        Application app1;
        app1 = appIter.next();
        app1.resume(as);
/*the high-level resume operator takes care of discovering the best devices in the Active Space on which to
resume the model, presentation, controller, adapter and coordinator components of the Application. It then
starts the components on those devices. Here, the dimension of utility to use for finding the "best"
components and devices is defined by space-level policies or user preferences*/
}
```

7. Implementation and Experiences

We have implemented the Olympus programming framework as part of Gaia middleware.. Operators in the language are implemented as libraries. Policies are specified in XSB Prolog [10]. Ontologies in OWL were developed using Protégé [11]. A Protégé plugin also offers web-based browsing of ontologies. This allows developers to look up various concepts and properties while developing their programs. Location-based reasoning, such as calculation of distance between two entities is done using the Gaia Location Middleware [12].

We have used the Olympus framework to develop a number of programs for Active Spaces. We found the framework did speed up development time since the discovery of appropriate entities and common operations were abstracted away from the developer. Programs developed using this framework were also more portable since they did not rely on specific resources available in or configurations of Active Spaces. Hence, we were able to deploy these programs rapidly in different spaces in our Computer Science building.

The use of ontologies greatly simplified development since developers were now aware of the different kinds of entities and their properties. The framework, though, is based on the assumption that all Active Spaces that use the framework share a common ontology. If a new space that uses a different ontology needs to be integrated, then translations or mappings have to be specified between the different ontologies.

Finally, although the framework does simplify the task of the developer, it takes away some control from the hands of the end-user. Many configurations decisions are made automatically, although user policies and preferences are taken into account. We are working on better interfaces that allow end-users to interact with the decision making process of the Olympus framework.

8. Related Work

Various programming models have been proposed for pervasive computing environments. The task-computing model [13] 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. It, however, does not have a utility model for choosing the best way of performing tasks. It also does not use policies to guide the execution of tasks. The operator graph model [14] uses a programming model where services to be composed are specified as descriptions and interactions among services are defined using operators. It also does not have any mechanisms for comparing different ways of composing the services.

Various discovery mechanisms such as Jini Lookup, the CORBA Trader, UDDI, etc have been used for

discovering entities that satisfy certain constraints. However, most of these mechanisms specify constraints based on name-value pairs. They don't have the notion of semantic similarity of classes. The discovery process is also not integrated with policies. Also, none of them make use of a flexible utility function for choosing "best" candidates.

9. Conclusion and Future Work

In this paper, we have presented a high-level programming framework for pervasive computing environments. The framework relieves developers from the task of discovering appropriate entities for performing a task. It also provides developers with abstractions for common Active Space operations. Finally, it provides a way of integrating policies and a utility model into the discovery process.

In the future, we plan to implement a version of the framework as a scripting language as well. This would allow developers to write scripts quickly for performing various tasks and test time immediately without having to compile them. In our current prototype, administrators, developers and users have to specify policies in Prolog. We are looking at other ways for specifying rules and policies. Another area of interest is more expressive utility functions. In particular, we would like to experiment with attaching weights to different utility dimensions, so that the net utility of an instance is a linear combination of the utility in different dimension. We are also working on using machine learning approaches to learn certain kinds of rules, especially user preferences such which kinds of devices and applications they prefer to use for different kinds of tasks. Finally, we are also exploring different ways of automatically composing several operators to achieve a high-level goal. One promising approach in this direction is the AI planning.

10. References

[1] M. Roman, C. K. Hess, R. Cerqueira, R. H. Campbell, and K. Narhstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," *IEEE Pervasive Computing Magazine*, vol. 1, pp. 74-83, 2002.

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

[3] OMG, "CORBA, Architecture and Specification," Common Object Request Broker Architecture (CORBA) 1998.

[4] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, pp. 123-169, 1998.

[5] M. Dean, D. Connolly, F. v. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL web ontology language 1.0 reference," http://www.w3.org/TR/owl-ref/, 2002.

[6] 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.

[7] M. Roman, H. Ho, and R. H. Campbell, "Application Mobility in Active Spaces," presented at 1st International Conference on Mobile and Ubiquitous Multimedia, Oulu, Finland, 2002.

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

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

[10] "XSB Prolog." http://xsb.sourceforge.net.

[11] N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen, "Creating Semantic Web Contents with Protege-2000," *IEEE Intelligent Systems*, vol. 16, pp. 60-71, 2001.

[12] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R. Campbell, and M. D. Mickunas, "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications," In *ACM/IFIP/USENIX 5'th International Middleware Conference,* Toronto, Canada, Oct 18-22, 2004

[13] Z. Song, Y. Labrou, and R. Masuoka, "Dynamic Service Discovery and Management in Task Computing," presented at First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04), 2004.

[14] 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.