

Mobile Polymorphic Applications in Ubiquitous Computing Environments

Anand Ranganathan, Shiva Chetan and Roy Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
{ranganat, chetan, rhc}@uiuc.edu

Abstract

Ubiquitous Computing envisions an environment where physical and digital devices are seamlessly integrated. Users can access their applications and data anywhere in the environment. Applications are not bound to any single device and can migrate with the user to different environments. Therefore, application mobility is an important aspect of ubiquitous computing.

In this paper, we consider the problem of migrating applications across different ubiquitous computing environments (i.e. across different rooms, buildings or even cities). Migration is a tough problem because different environments have different resources (devices or services) available. The context of the environments may be different as well. Hence, mobile applications must adapt to changing contexts and resource availabilities as they migrate from one environment to the next. We introduce the notion of polymorphic applications, where applications can change their structure in order to adapt to different environments. While the structure of polymorphic applications can change during migration, the functionality and the state of the application are preserved as far as possible. This enables users to perform the same tasks as they move from one environment to the next, seamlessly. We make use of ontologies to ensure that the initial and final structures of a migrating application are semantically similar in terms of functionality and behavior. This paper describes our framework for enabling mobile polymorphic applications.

1. Introduction

Ubiquitous Computing promotes a physical environment that seamlessly assimilates digital devices. It provides a framework for users to access their applications and data anywhere in the environment. Our notion of a ubiquitous environment is a physically bounded space of physical and digital devices such as lights, cameras, handheld devices and desktop computers. We call this space an *Active Space*.

An Active Space consists of various entities including users, applications, services and devices.

Mark Weiser [2] envisioned ubiquitous computing as a system in which applications move with the user. The traditional notion of application-to-computer association has to be revised to application-to-user association. Applications should automatically migrate to where the user is and adapt to the resources present in the user's environment.

Previous research in application migration [3, 4, 5] supported adaptation based on changes in device and network topology. They provided little or no support for adaptation based on context information. Further, application structure was not altered during adaptation due to changing resource availabilities. Our migration framework enables structural adaptation by automatically decomposing an application into smaller components, each of which can independently adapt to the new environment, and then recombining the adapted components. The functionality and the behavior of the application are preserved during this process, so that the user can continue performing the same tasks as he moves around.

In this paper, we present our notion of application polymorphism, wherein an application can exist in different forms during its lifetime. This property allows a mobile application to change its structure in order to adapt to different environments. The structure of an application in an Active Space is based on the Model-View-Controller framework [6]. An application consists of input (controller), output (view) and logic (model) components. For example, a presentation application (PowerPoint) can be split into (i) one model component that runs on a laptop and maintains the high-level state of the application such as the name of the file being presented and the slide number; (ii) one controller component that runs on a desktop and allows the user to navigate the slides and (iii) many view components that run on several wall-mounted displays. When an application migrates to another Active Space, each component of the application is adapted independently. In addition, the number of instances of the components (i.e. component cardinality) can

change based on the availability of resources in the Active Space.

Our framework allows 3 kinds of application adaptation:

1. Change in the type of components – for example, a PowerPoint view can be replaced by an Acrobat Reader view with suitable transcoding since an Acrobat Reader can also be used to give a slide-show.
2. Change in the number of components – for example, the number of views of an application (like Acrobat Reader) can be increased if there are many wall-mounted displays that can be used to show the slides in the new environment.
3. Change in the devices on which these components run – for example, a controller for the slide-show may initially have been running on a laptop. In the new space, the controller may be instantiated on a PDA due to unavailability of a similar laptop.

While applications can adapt in the above manners, they should still allow users to perform the original intended tasks. For example, if the original application used WinAmp to play an mp3 file, then the new application structure should still allow the user to listen to music (using, e.g., some music player and speaker).

In order to achieve task continuity, we have developed the notion of semantic similarity of application components. Our view of the semantics of an application is based on the tasks the application component 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 example, a 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.

Semantic similarity between components is determined with the help of ontologies that describe the different kinds of components. Ontologies define a hierarchy of entities in the space based on the kinds of tasks they help users perform. In this paper, we present the algorithm we use to determine how similar two components are, based on the ontological hierarchy. The ontologies also have information on the kinds of devices required to run them. If there are multiple devices available in the new space to run a component,

context as well as user preferences are used to choose the best device(s) to run the component on. Context information and device availability also dictate whether to increase or decrease the number of view or controller components in the application. Context information includes the location of the user in the room, the location of devices, whether the devices already have some application running on them, the presence of other people in the room, the current activity of the user and so on.

We have developed a prototype system that supports migrating polymorphic applications. This system has been built on top of Gaia [1], our meta-operating system that manages various physical and digital entities in an Active Space.

2. Scenario

Sal is preparing for a presentation to a group of students. She reviews her PowerPoint slides in her “smart” office before the presentation. The slide-show application consists of a *PPTModel* component that stores the high-level state of the application (name of file and slide number), a *WindowsSlideShowController* GUI that is used to change slides and a *PowerPointViewer* component that displays the slides using Microsoft PowerPoint. The *PPTModel* runs on her laptop that she used to design the presentation, the *WindowsSlideShowController* runs on her laptop as well, and the *PowerPointViewer* displays slides on a wall-mounted display.

After reviewing the slides, Sal suspends her application. The smart office stores the application state and structure in a file. Sal stores this file in her USB flash drive. Upon entering the smart presentation room, Sal plugs in her USB drive into an available computer and provides the file to the smart room. The room observes that the slides are in PowerPoint form but does not find a PowerPoint application in the room (since all machines in the presentation room run Linux). It, however, discovers that it can use *Acrobat Reader* with suitable transcoding to display the slides in the room. Context-sensitive configuration rules in the room dictate that slides should be displayed on all four screens in the room. So multiple instances of the Acrobat Reader are spawned and slides are displayed on all screens in the room. Sal had earlier indicated a preference for having the controller on her PDA (which runs Linux) when she is giving a presentation. So, the *LinuxSlideShowController* is started on her PDA.

The above scenario showed how our system adapted the structure of Sal’s slideshow application based on the resources available in and the context of the new space. The application view is *morphed* from one that uses PowerPoint to display the slides on a single screen to one that uses Acrobat Reader on Linux

to display the slides on four screens. The application controller is *morphed* from a *WindowsSlideShowController* running on a laptop to a *LinuxSlideShowController* running on a PDA. However, the core functionality of the application (i.e. it displays slides) is preserved and Sal can continue to display slides even though the new room has different resources.

3. Features of the System

Today, migrating applications across different environments is a cumbersome task for users. It involves several steps such as saving the application, transporting the saved application, looking for appropriate devices in the new space and restarting it there. Our system allows automatic migration of applications and thus, enhances user experience.

Our system supports the partitioning of an application across different machines. This allows us to decompose an application into input, output and logic components and adapt each of them separately while the application is being migrated. The functional partitioning of the application increases the adaptability of the application and makes use of the large number of devices present in ubiquitous computing environments.

One of the main problems while migrating applications across spaces is the difference in the kinds of devices that may be available in the two spaces. Each kind of application component (input, output or logic) can run on only certain kinds of devices. For example, a PowerPoint output component can only run on a desktop or a laptop having a display and running Windows. The new space may not have such a device – in that case, the application should be suitably modified to run in whatever devices the space has (such as a Linux machine or a handheld device). Our system adapts applications by discovering equivalent application components that can perform the same task and then discovering appropriate devices to run the components.

While adapting applications, our system ensures that the initial and final application component types are semantically similar. Two components are semantic similar if they allow users to perform similar tasks. Hence, one component may be substituted by the other. We use ontologies, written in DAML+OIL, to describe the types of application components and to check how similar two different components are.

Context and user preferences are also used while choosing appropriate devices to run application components. For example, if there are multiple displays in a room, then the display closest to the user could be chosen for the application. If, however, the display is already being used by another application, then it may

not be appropriate. User preferences and context rules allow removing inappropriate devices from consideration and picking suitable ones.

4. Application Polymorphism

The application framework [3] in Gaia is an extension of the Model-View-Controller framework [6]. It defines five components: model, presentation (generalization of view), adapter, controller and coordinator. The framework is shown in Figure 1.

The model implements the logic of the application and exports an interface to access and manage the application's state [3]. 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. More details of the framework can be found in [3].

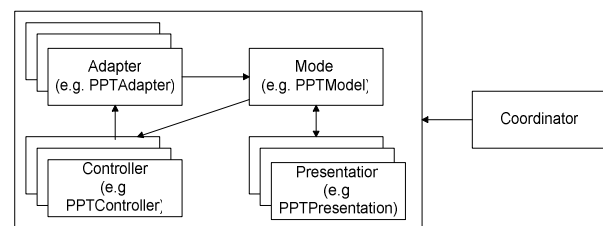


Figure 1. Application Framework and the components of an example slide-show application (in parentheses)

The structure of an application is described by the type and cardinality of the components and the devices to which they are mapped. A change in any of these characteristics is regarded as *structural adaptation*. A change in the type of the components is regarded as a *type adaptation* and a change in the cardinality as a *cardinality adaptation*. Finally, *device mapping* involves binding application components to specific devices in the space.

In type adaptation, an application can change the type of controller or presentation component being used. In the scenario, *LinuxSlideShowController* replaced *WindowsSlideShowController* as the controller component of the slide-show application. In cardinality adaptation, the number of controller and presentation components can change based on the context of the new Active Space, availability of devices and user preferences. For example, a slide-show presentation may be replicated to display slides on all plasma displays in a conference room Active Space.

The different types of adaptations allow *application polymorphism* - the property by which an

application can exist in different structural forms during its lifetime while preserving the semantics of the application. The semantics of an Active Space application is based on the tasks that the application allows the user to perform. Polymorphic applications can adapt their structure to suit the resources present in a ubiquitous computing environment, yet allowing the user to perform the original intended tasks.

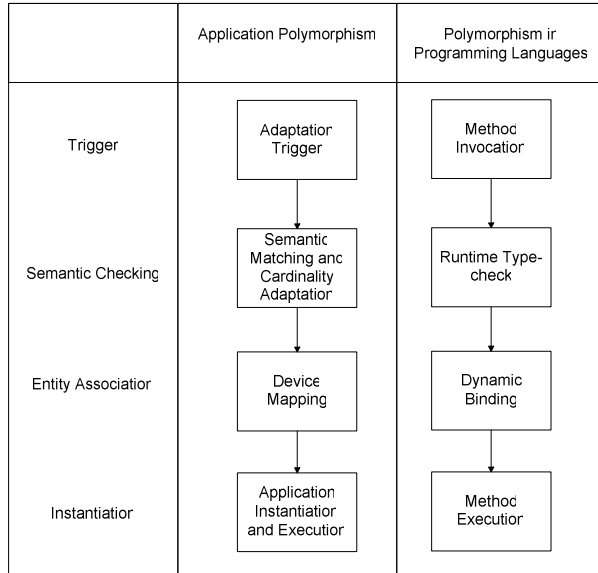


Figure 2. Analogy between Polymorphism in Programming Languages and Application Polymorphism

We draw a rough analogy between polymorphism in programming languages and application polymorphism in Figure 2. Application adaptation is triggered by various conditions, such as application migration or when a device on which an application component is running fails. This triggering is analogous to a polymorphic method invocation in an object-oriented language. Semantically matching components are discovered using ontologies and adapted to the right structure based on context. This is analogous to runtime type checking in a polymorphic method call where the appropriate method is discovered based on the runtime type of the object invoking the method. The discovered components are mapped to devices analogous to dynamic binding of method calls. Finally, the application is instantiated and executed on the mapped devices analogous to a method execution.

5. Migration Algorithm

The steps involved in migrating an application are:

1. The user triggers the migration process through a GUI.
2. The structure of the migrating application as well as its current state is stored in an Application Customized Description (ACD) file.
3. The ACD of the application is sent to the new space.
4. A Migration Service in the new space takes the old ACD of the application or its file handle and generates a new ACD for the application, after performing type and cardinality adaptation and device mapping. It involves the following steps:
 - a. The Migration Service queries the Ontology Server in the new space for classes of presentations and controllers that are semantically similar to the components listed in the old ACD. The Ontology Server returns an ordered list of classes that are semantically similar to the old presentations and controllers.
 - b. The Migration Service filters the list of classes returned depending on whether a path exists in its transcoder graph between the data formats.
 - c. For each candidate component class, the Migration Service gets classes of devices that can host the component class from the Ontology Server.
 - d. The Migration Service then queries the Space Repository to get instances of the classes of devices (obtained from the previous step) that are available in the new space. The Space Repository is a database containing information about all devices, components, services and users in the Active Space.
 - e. For each presentation and controller present in the application, the Migration Service decides the cardinality and the devices on which the components must be instantiated using rules involving the context of the new space and preferences of the user. Alternatively, the user can also specify the devices to which components should be mapped using a GUI.
5. Once the Migration Service arrives at a new application structure, it instantiates this application in the new Space

6. Semantic Similarity of application components

Before we go into the details of the algorithm, we define our notion of semantic similarity between concepts in general, and different application components in particular. Semantic similarity is based on the functionality and behavior of the components

inferred using ontologies that describe the different types of components and their properties.

Each entity in the Active Space has a DAML+OIL file (one of the standard formats of the Semantic Web) associated with it that describes its properties. In particular, all application components have a DAML+OIL file describing their properties. This file 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 ontologies also create a hierarchy (or a taxonomy) of all the entities in a space. This hierarchy is based on functionality and behavior. A portion of this hierarchy is shown in the Figure 3a. The figure shows some of the kinds of entities in the space and in particular, shows some kinds of application presentation components. 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.

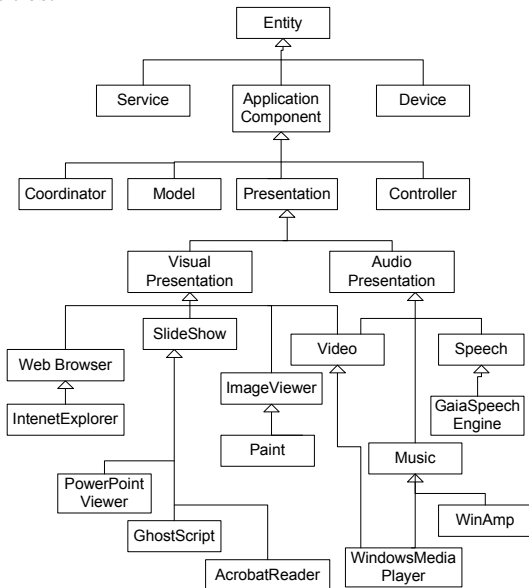


Figure 3a. Presentation hierarchy in Gaia

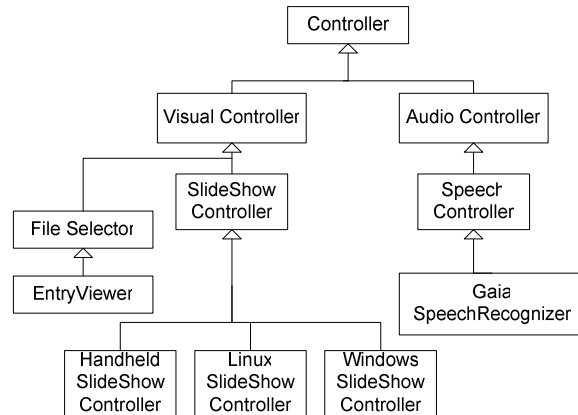


Figure 3b. Controller hierarchy in Gaia

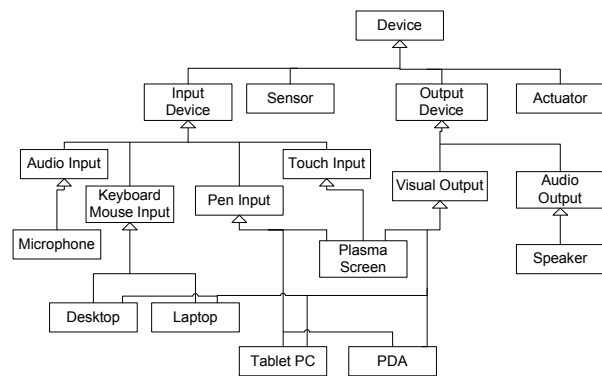


Figure 3c. Device hierarchy in Gaia

Similar hierarchies exist for different types of controllers (Figure 3b), devices (Figure 3c) and operating systems. Besides, 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.

requiresDevice : ApplicationComponent → 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 such relation is *requiresOS* that maps application components to operating systems, e.g.
requiresOS(PowerPointViewer) = Windows

6.1. Ontology Server

An Ontology Server maintains all the ontologies in Gaia. Other entities in Gaia contact the Ontology

Server to get semantic descriptions of entities in the space as well as to find concepts (or classes) that are semantically similar to other concepts. The Ontology Server uses the FaCT Reasoner[17] to reason about concepts in the system (for example, to perform logical queries like subsumption, classification and satisfiability of concepts – which are required for finding semantically similar classes).

6.2. Semantic Matching of concepts

The process of finding semantically similar concepts makes use of the ontological hierarchy. We implemented an adapted version of the algorithm presented in [14, 15]. 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* gives the following classes:

Similarity-level 0 : PowerPointViewer
 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.

7. Triggering Migration

The graphical interface to the Migration Service is shown in Figure 4. Users can select one or more applications to migrate and choose if they want the adaptation to be manual or automatic.

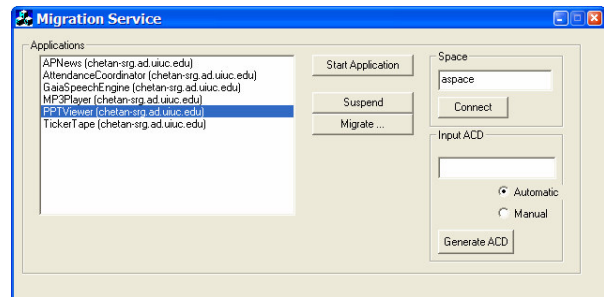


Figure 4. Interface of Migration Service

8. Capturing Structure and State of an Application

When an application is suspended, a “snapshot” of the application is stored in an ACD. The snapshot contains the description of the structure of the application and application state stored in the model. An ACD can be thought of as analogous to an executable file of a computer. The ACD of the slide-show application is shown in Figure 5.

```
Application = {
  Model =
  { {   ClassName = "PPTModel",
        Hosts = {{ "desktop1.aspace.cs.uiuc.edu" }}, } },
  Presentation =
  { {   ClassName = "PowerPointViewer",
        Hosts = {{ "plasma.aspace.cs.uiuc.edu",
                  {"touchscreen.aspace.cs.uiuc.edu"} } },
  Controller =
  { {   ClassName = "WindowsSlideShowController",
        Hosts = {{ "pda.aspace.cs.uiuc.edu" }}, } },
  Coordinator =
  { {   ClassName = "Coordinator",
        Hosts = {{ "desktop1.aspace.cs.uiuc.edu" }}, } },
  State =
  { {   Path = "/aspace/bill/ss_state.dat", } }
}
```

Figure 5. ACD of a slide-show application

The ACD and the state are stored in the Gaia Context File System (CFS) [7]. The CFS provides a unified file system that is shared by all devices in an Active Space. The CFS also provides interfaces to access the file system from outside an Active Space. The *Path* attribute specifies the path of the file that stores the state of the model. In our implementation, we store the high-level state of an application. The high-level state of a slide-show application contains the presentation file and slide number. The data that constitutes the high-level state of an application model is specified by the developer. The high-level state is stored only for convenience and the system can be extended easily to store the memory snapshot of the model.

9. ACD transfer

The Migration Service provides a mechanism to specify the Active Space to which the application is to be migrated. The Migration Service contacts its counterpart in the target space and sends it the ACD. Alternatively, the ACD can be transferred manually by storing the file on a handheld or wearable device and loading it to the Migration Service of the target space (again using the GUI in Figure 4). A third alternative is to use a file reference since the ACD is a file in the Context File System.

10. Application Adaptation

When an application migrates to another Active Space, the ACD of the suspended application is fed to the Migration Service in the Active Space to generate an ACD customized for the new space (Figure. 6). The Migration Service supports two adaptation modes: manual and automatic. Type adaptation and device discovery are common to both modes. In the manual mode, users interact with a GUI tool to select a mapping of components to devices while in the automatic mode the Migration Service does this automatically using the context information of the space. Once the application structure is determined the ACD for the new space is generated.

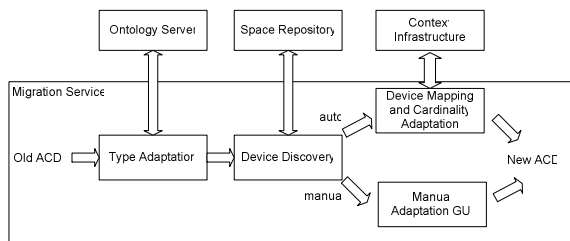


Figure 6. ACD generation during migration

10.1. Type Adaptation based on Ontologies

Type adaptation refers to a change in the type of components used when an application is adapted in an Active Space. During type adaptation, the controller and the presentation components may be replaced by semantically similar components if the original components cannot be supported.

For each presentation and controller component, the Migration Service gets the set of all compatible component classes and their level of similarity to the original component by querying the ontology server. It also checks to see if there exists a path in the transcoder graph linking the two component classes. The Context File System [7] of Gaia supports automatic transcoding of data by linking various transcoders. This work reuses some of the earlier works done in transforming data to different formats using graphs such as [16]. If no path exists in the graph, then the component class is discarded.

10.2. Device Discovery

The ontology server also provides information on which classes of devices can host various classes of components. The Space Repository (which is a registry of all entities in a Space) is then queried for these classes of devices. Figure 7 gives an example of a component-device table after device discovery.

Controller Component:

Similarity-level	Component Class	Device
0	WindowsSlideShowController	
2	LinuxSlideShowController	pda1, pda2, pda3
3	EntryViewer	laptop1
4	GaiaSpeechRecognizer	mic-pc

Presentation Component:

Similarity-level	Component Class	Device
0	PowerPointViewer	
2	AcrobatReader	winpc1, winpc2, plasma1, plasma2, plasma3, plasma4
2	GhostScript	plasma1, plasma2, plasma3, plasma4, linux1, laptop1
3	InternetExplorer	tabletpc1, tabletpc2
3	Paint	pda1, pda2, pda3, winpc1, winpc2
3	WindowsMediaPlayer	speaker1, speaker2
4	WinAmp	laptop1
4	GaiaSpeechEngine	speaker1, speaker2

Figure 7. Component-device table of slide-show application after device discovery

10.3. Device Mapping and Cardinality Adaptation

The Migration Service filters the set of possible devices in the component-device table using context-sensitive rules. The rules, specified in Prolog, could prevent certain devices from being used (for example, if a display is already being used entirely by another application, then it can't be used to display something else). If no device can host the component class, then it tries component classes of the next similarity level. There are two types of rules – space-level rules (that are set by a system administrator for a certain space), and user-level rules that specify an end-user's preferences. Both these two types of rules are used to decide appropriate devices, though the space-level rules have higher priority. A subset of such rules is shown below:

```
canHostPresentation(x, y) :- isSlideShow(x),
isPlasma(y), not(runningVisComp(y)).
/*i.e. a Plasma Screen that is not displaying another
visual component can be used to host a slide show*/
isSlideShow(x) :- subclass(x, 'SlideShow').
/* a slideshow application is one that is a subclass of
SlideShow as defined in the ontology*/
isPlasma(y) :- instanceOf('PlasmaScreen', y)
```

```
canHostPresentation(x,y) :- isWebBrowser(x),
isTabPC(y), not(runningVisComp(y)).
/*i.e. a Tablet PC that is not displaying another visual
component can be used to host a web browser*/
isWebBrowser(x) :- subclass(x, 'WebBrowser').
isTabPC(y) :- instanceOf('TabletPC', y).
```

There are also rules that specify the cardinality of a component in the application. For example, a rule can specify that all available devices in the room should be used for hosting the component, or a certain number of them should be used. For example,

```
presentationCardinality(X, Y, all) :-
roomActivity(presentation), userRole(Presenter),
isPlasma(Y).
/*All devices of type Y are to be used for the
presentation of component X*/.
```

The Migration Service, thus, finds one or more devices to host presentations and controllers. If there are many candidate solutions, it chooses one randomly. We assume that presentations and controllers, and the devices they are hosted on, are independent, i.e. the choice of a certain presentation component and a certain set of devices for it does not influence the choice of controller class or controller devices and vice-versa. This assumption is based on the fact that the application partitioning framework splits an application into functionally independent components.

Once the Migration Service deduces the complete application structure, it instantiates the components of the application. The model is initialized to the state captured when the application was suspended. The Coordinator composes the application components and then resumes the application.

10.4. Manual Adaptation

In the manual adaptation mode, the user interacts with the Migration Service using a GUI to choose the presentation and controller devices for an application. The GUI lists the component-device table and allows users to pick the components and devices. These components then form the new application structure.

11. Implementation and Experiences

We have built a prototype system and have used it to migrate applications between various rooms in our Computer Science building. Each room is a different Active Space. One of the rooms is a lab that is equipped with 5 plasma screens, 2 tablet PCs and various sensors for detecting location based on RF badges. The other rooms are offices and conference rooms. An office typically has one or two desktops and a laptop. Some offices only have Linux machines while others have Windows and Linux machines. A conference room has a number of laptops and a single computer connected to projector. PDAs can be used in all rooms. We have used our system to migrate applications like slideshow applications, music playing applications and web browsers between various rooms. Different spaces have different kinds of devices and also have different types of application components installed (for example, some spaces have WinAmp installed while others do not). The system adapts the applications as they migrate depending on the application components and kinds of devices available.

Our system reduces the number of tasks users have to perform to migrate applications. Earlier, users had to save applications explicitly, carry the saved files physically or transfer it using ftp, find appropriate devices in the new room and restart it. Also, some of the state (such as the slide number of a presentation or the exact location within a song) was lost. Our system allows users to just choose applications to migrate to a new space and the migration happens automatically. At the same time, our system allows users to override the system choices or manually specify devices to map different application components to. Thus, users still have control over the migration process, which is an important element in any proactive system.

In terms of performance, the time taken to migrate an application consists of two elements – the time

taken to save the application state and structure as an ACD, and the time taken to generate the new ACD in the new space and restart the application. We found that the first operation took an average of 0.044 sec and the second operation took 7.45 sec. The experiments were performed on Pentium 3, 790 Mhz, 256 MB machines. These numbers indicate that the migration process is fast enough to make it practical.

12. Related Work

Several research projects have contributed to automatic adaptation during application migration. one.world [5] supports application migration and adaptation using a virtual machine environment. iMASH [10] focuses on data conversion to support mobility and adaptation of applications. Aura [11] uses a distributed file system to transfer information for application migration. Adaptation in Aura is facilitated by two application abstractions: Suppliers and Connectors. The main difference between these systems and Gaia is that they assume a single-device application model while Gaia uses a multi-device application model. Further, application adaptation in Gaia is based on ontological classification and it uses context information as well.

I-Crafter [8] and PIMA [9] support adaptation during mobility. The adaptation mechanism in I-Crafter uses context information for generating appropriate user interfaces. PIMA also proposes adaptation based on context information. Our system, however, also provides a framework for decomposing an application during adaptation. Further, we use ontology classification for adaptation while the above projects use only context information.

Metaglué [18] provides mechanisms to bind together large groups of software agents. Metaglué is an extension to the Java programming language and provides linguistic primitives to specify computational requirements in pervasive environments. The focus of Metaglué is on bridging the semantic gap that exists between heterogeneous software components. The focus of our work is to find substitutable software components and semantic bridging of data incompatibilities is done by our Context File System using service composition techniques.

There has been a lot of work on semantic matchmaking using ontologies[14, 15], especially for web services[12, 13]. We have borrowed ideas from some of these works, especially in coming up with degrees of match, and deciding how semantically similar two application components are.

The Task Computing project [19] uses ontologies for bridging semantic gaps in pervasive environments. The focus of the project is to enable service

composition using ontologies. The project does not define a framework for application decomposition and does not address issues in migrating applications across pervasive environments. The focus of our work is to enable a seamless user-experience when users move around in a pervasive environment.

13. Conclusions and Future Work

In this paper, we introduce a class of applications – called polymorphic applications - that can exist in different forms during their lifetime. Polymorphic application can change their structure to suit the requirements of a ubiquitous environment. We have developed a system that allows migration of polymorphic applications across different environments. Our system uses ontologies and context information to decide how best to adapt migrating applications to enable a seamless user experience.

As part of future work, we plan to work on automatic triggering of migration based on location information. We also want to experiment with machine learning to decide how best to adapt applications based on the way users reconfigure migrated applications.

References

- [1] Román, M. et al., “Gaia: A Middleware Infrastructure to Enable Active Spaces”, *IEEE Pervasive Computing*, Oct-Dec 2002, pp. 74-83.
- [2] Mark Weiser, “Hot Topics: Ubiquitous Computing”, *IEEE Computer*, October 1993.
- [3] Roman, M. et al., “Application Mobility in Active Spaces”, *1st International Conference on Mobile and Ubiquitous Multimedia*, Oulu, Finland, Dec 11-13, 2002.
- [4] Boyd, T. and Dasgupta, P., “Process Migration: A Generalized Approach using a Virtualizing Operating System”, *ICDCS 2002*, Vienna, July 2002.
- [5] Grimm, R., “System support for pervasive applications”. Ph.D. thesis, University of Washington, December 2002.
- [6] Krasner G E and Pope S T, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system”, *Journal of Object-Oriented-Programming* pp. 26-49.
- [7] Christopher K. Hess, “The Design and Implementation of a Context-Aware File System for Ubiquitous Computing Applications”, Ph.D Thesis, 2003.
- [8] Ponkanti S R, et.al., “ICrafter: A service framework for ubiquitous computing environments”, *Proc. Ubicomp 2001*, 2001.
- [9] Banavar G et.al., “An application model for pervasive computing”, *Proc. 6th ACM MOBICOM*, Boston MA
- [10] Phan T et.al., “A new twist on mobile computing: Two-way interactive session transfer”, *Proc. Workshop on Internet Applications (WIAPP 2001)*, San Jose, 2001.
- [11] Sousa J P and Garlan D, “Aura: An architectural framework for user mobility in ubiquitous computing

- environments”, *Proc IEEE/IFIP Conference on Software Architecture*, Montreal, 2002.
- [12] Massimo, P., et al, “Semantic Matching of Web Service Capabilities,” *First International Semantic Web Conference*, Sardinia, 2002
- [13] Sycara, K. et al, “Dynamic Service Matchmaking Among Agents in Open Information Environments,” *ACM SIGMOD Record*, vol. 28, no. 1, 1999, pp. 47-53.
- [14] Li, L. and Horrocks, I., “A software framework for matchmaking based on semantic web technology,” *In Proc. of the twelfth international conference on World Wide Web*, 2003, pp. 331-339.
- [15]Gonzalez-Castillo, J. et al. “Description Logics for Matchmaking Services”, HP Laboratories Bristol, Bristol HPL-2001-265, 2002.
- [16] Raman, B. et al, “Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network”, WMCSA 2000, California, USA.
- [17] Horrocks, I. and Patel-Schneider, P.F. “The FaCT system”. In *International Conference, Tableaux’98*.
- [18] Coen M, Phillips B, Warshawsky N, Weisman L, Peters S and Finin P (1999) “Meeting the computational needs of intelligent environments: The metagluue system”, *Proc. MANSE’99*, Dublin, Ireland
- [19] Masuoka R, Parsia B and Labrou Y, “Task Computing - the Semantic Web meets Pervasive Computing”, *2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, USA, 20-23 October 2003.