A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments[1]

Manuel Roman and Roy H. Campbell
mroman1, rhc@[cs.uiuc.edu]
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Report No. UIUCDCS-R-2002-2284 UILU-ENG-2002-1728

July, 2002

[1] This research is supported by a grant from the National Science Foundation, NSF 0086094, NSF 98-70736, and NSF 99-72884 CISE.

A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments

Abstract

Ubiquitous computing challenges the conventional notion of a user logged into a personal computing device, whether it is a desktop, a laptop, or a digital assistant. When the physical environment of a user contains hundreds of networked computer devices each of which may be used to support one or more user applications, the notion of personal computing becomes inadequate. Further, when a group of users share such a physical environment, new forms of sharing, cooperation and collaboration are possible and mobile users may constantly change the computers with which they interact. We present in this paper an application framework that provides mechanisms to construct, run or adapt existing applications to ubiquitous computing environments. The framework binds applications to users, uses multiple devices simultaneously, and exploits resource management within the users' environment that reacts to context and mobility. Our research contributes to application mobility, partitioning and adaptation within device rich environments, and uses context-awareness to focus the resources of ubiquitous computing environments on the needs of users.

Contents

Introduction

Future ubiquitous computing will surround users with a comfortable and convenient information environment that merges physical and computational infrastructures into an integrated habitat. Context-awareness should accommodate the habitat to the user preferences and tasks, group activities, and the nature of the physical space. We term this dynamic and computational rich habitat an active space. Within the space, users will interact with flexible applications that may move with the user, may define the function of the habitat, or collaborate with remote applications. The research described in this paper builds on experiments with applications conducted in an active space (Figure 1) and contributes to ubiquitous computing with the notion of user-centric, resource-aware, context-sensitive, multi-device, and mobile applications.

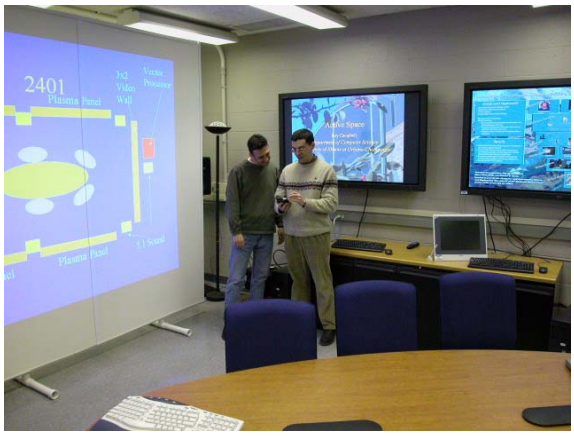

Figure 1. Experimental Active Space

The active space consists of the Gaia middleware OS infrastructure operating a distributed system composed of four 61" wall-mounted plasma displays, a video wall, 5.1 audio system, touch screens, IR beacons, badge detectors, and wireless and wired networks connecting 15 Pentium-4 PCs running Windows 2000 and Windows CE based Compaq iPaq PDAs. Gaia supplies services including event delivery, entity presence detection (devices, users, and services), context notification, a space repository to store information about entities present in the space, and a context-aware file system [1].

The application experiments examine how to construct applications that use multiple devices simultaneously, take advantage of resources contained in the user habitat, exploit context information (e.g., location, mood, and social activity), benefit from automatic data transformation and can alter their composition dynamically (e.g., attaching and detaching components) to adapt to changes in the habitat, and move with the users to different active spaces.

The proposed application framework leverages previous work on automatic application customization for heterogeneous devices (individual devices)[2, 3], static partitioning of applications into multiple devices [4], and application development for group collaboration in active meeting rooms[5]. It extends this work to accommodate the requirements of active spaces, by customizing applications to use multiple heterogeneous devices, altering the application partitioning dynamically, and constructing collaborative and personal applications for arbitrary active spaces.

The problems we focus in this paper are: (1) defining an application infrastructure that can accommodate the requirements of active spaces including dynamically changing the cardinality, location, and quality of input, output, and processing devices used by an application; (2) providing a specialization mechanism that allows defining applications requirements generically and automatically mapping them to the resources present in a particular active space; and (3) building policies to define adaptation rules for applications.

The paper continues with a description of the issues we believe are key for ubiquitous applications (section 2), a description of the application framework (section 3) including information about the infrastructure (section 3.1) the specialization process (section 3.2), and the application policies (section 3.3). Section 4 presents an example of an application we have built using the framework. We present related work in section 5, and conclude the paper and present our future work in section 6.

Ubiquitous Applications Key Issues

This section presents five issues we consider essential for ubiquitous applications. These five issues are the cornerstones of the proposed application framework.

Resource-Awareness

Users in ubiquitous computing scenarios are surrounded by hundreds of devices including sensors, displays, and CPUs. In order for applications to exploit these resources, they must be aware of existing resources; the capabilities, availability, and cardinality of these resources. An active meeting room, for example, is aware of existing devices (e.g., plasma displays, projectors, servers, notebooks, PDAs, and sensors), services (e.g., light control, temperature control, and audio control), applications (e.g., collaborative document editor, and slide show manager), and people present in the meeting room, therefore allowing applications to exploit the resources.

Context-Sensitivity

Context is one of the most important properties in ubiquitous computing [6] and affects the behavior of applications. We identify at least three application aspects that are altered by context: data, composition, and logic. Applications require transforming the format of the output data to accommodate changes in the context. For example, an e-mail application using a large monitor to display the e-mail's text, may require transforming the text to speech when the user moves to a new location where only speakers are available. Context also affects the internal composition of the application - number of application components and their composition rules. A music application may use a user's laptop to play the music if there are other people present in the room; or may use the audio system of the room, the displays (to present the list of songs), and the room's speech recognition system to control the application when the user is alone. Finally the logic of the application may also be affected by the context situation. For example, a news broadcasting application may select different types of news depending on who is in the room, the time of the day, or the mood of the users.

Multi-Device

In an environment where users are surrounded by hundreds of devices, the notion of logging into a single device becomes inappropriate. Users log into the active space (either implicitly or explicitly) and can take benefit of any entity contained in the space, as long as certain security and availability policies apply. This "post-pc" scenario requires a new model for application construction that allows partitioning applications into different devices as required by users and their associated context (e.g., time of the day, location, current task, and number of people). Application partitioning allows distributing functional aspects of an application (e.g., application logic, output, and input) across different devices. Remote terminal systems (such as X-Windows [7]) allow redirecting the application output and input to different devices. However, they do not provide support to redirect the application output to one device and the input to another device. And for the same application, it is not possible to redirect multiple outputs to different devices. The type of application partitioning we seek is conceptually similar to the one proposed by [4], and provides fine grained control to choose a target device for each individual application functional aspect, as well as support for altering the application partitioning at run-time.

The application partitioning must be: (1) dynamic, so it may vary at run-time according to changes in the active space (e.g., new devices introduced in the space, or new people entering the space), and (2) reliable, in such a way that guarantees application integrity even when the application is distributed across different devices.

For example, a calendar application running in an active office may use a plasma display to present the appointments for the week, a handheld to display the appointments for the day, and may use a desktop PC to enter data. However, the same calendar running in a vehicle may use the sound system to broadcast information about the next appointment, and use an input sensor based on speech recognition to query the calendar or to enter and delete data.

User-Centrism

Environmental-awareness and the multi-device approach convey a third essential property: user-centrism. To accommodate application partitioning into multiple devices that vary over time, we bind applications to users and map the applications to the resources present in the space.

Abowd et. al.[8] use the term "everyday computing" to denote the type of applications associated with users that do not have a clearly defined beginning and end. Users may start these applications and use them for

several days, months, or even years. Applications may be periodically suspended and resumed but not terminated. These applications are bound to users, and take benefit of the resources present in the users' environment. This application behavior requires applications to have permanent access to data (regardless of the users' context such as location), be able to access and take benefit of resources currently available, and be able to adapt to the current environment, either by changing their composition or by modifying the data they manipulate, or both.

User-Centrism requires applications to (1) move with the users, (2) adapt according to changes in the available resources (it may imply data format transformation, or internal application composition, or both), and (3) provide mechanisms to allow users to configure the application according to their personal preferences.

Mobility

Application partitioning and user-centrism require applications to be mobile. We consider two different types of mobility: intra-space mobility and inter-space mobility. Intra-space mobility is related to the migration of application components inside an active space and is the result of application partitioning among different devices. Intra-space mobility allows users and external services to move application components among different devices. Inter-space mobility concerns moving applications across different spaces, and is a consequence of the user-centrism (users are mobile by definition).

Description of the Application Framework

The application framework we propose models applications as a collection of distributed components, and reuses the Model-View-Controller[9]. The framework exploits resources present in the application environment, provides functionality to alter the application composition dynamically (i.e., number, type, and location of the application components, as well as data format they manipulate), is context-sensitive, implements a specialization mechanism that supports the creation of active space-independent applications, and provides policies to customize different aspects of the application including mobility. We have implemented the application framework leveraging the functionality exported by Gaia [10].

Infrastructure

The application framework defines four elements: model, presentation (generalization of View), controller, and coordinator. The model, presentation, and controller are the application base-level building blocks and are strictly related to the application domain functionality (e.g., music jukebox functionality). The coordinator manages the composition of the three base-level components and implements the application meta-level. It stores information about the composition of the application components and exports functionality to access and alter the component composition (e.g., attaching and detaching presentations and controllers and list current presentations). Figure 2 illustrates the framework, and Figure 3 lists the components' interfaces.

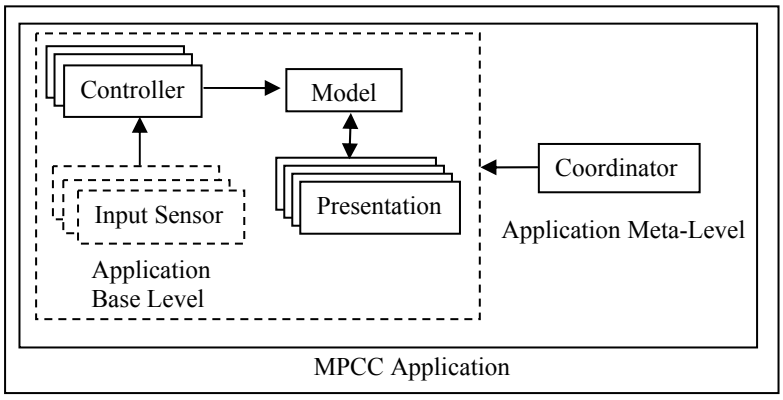

Figure 2. Model-Presentation-Controller-Coordinator Application Framework.

```
Interface Model {
  attachPresentation (in presentation)
  detachPresentation
  change ( in Any hint )
  saveState
  restoreState
  getCoordinator
  setCoordinator
}

interface Presentation {
  attach ( in model )
  detach
  notify ( in Any hint )
}

interface Controller {
  attach ( in model )
  detach
  defineEventMapping
  inputSensorEvent
}
```

```
interface InputSensor {
  attach ( in controller )
  detach
}

interface Coordinator {
  setModel ( in model )
  getModel
  registerPresentation ( in presentation )
  unregisterPresentation
  listPresentations
  registerController ( in controller )
  unregisterController
  listControllers
  registerInputSensor ( in inputSensor )
  unregisterInputSensor
  listInputSensors
  terminateApplication
  setOwner ( in owner )
  getOwner
}
```

Figure 3. Application Framework Component Interfaces

We have introduced four changes to the original MVC to accommodate the requirements for environmental-awareness, application partitioning, context-sensitivity, user-centrism, and mobility. First, we define a new component called presentation that models any output representation, not only graphical as proposed by the MVC view. Second, we generalize the definition of the MVC's input sensor (hardware device) to incorporate software components (e.g., context input sensor). Third, we introduce a new component called coordinator to explicitly manage the composition of the application components (meta-level). Finally, we generalize the input sensor time-sharing model defined by the MVC into a space-time-sharing model. According to MVC, all applications' views and controllers share the same input sensors (e.g., mouse and keyboard) and therefore the input sensors must be scheduled. Graspable interfaces [11] introduce the concept of space-sharing, where different input sensors are assigned to different functional aspects of the application, therefore avoiding the need for scheduling them. We combine both approaches into space-time-sharing to model the type of applications we consider. For example, a music application running in an active space uses a PDA to control the current song, and speech recognition to control the sound level (space-sharing), however the same space may host a calendar application that uses the PDA to browse appointments, and speech recognition to control the calendar's functionality (time-sharing). Space-time-sharing allows more than one controller and presentation to be active at the same time, which contrasts with MVC where only one controller-view pair can be active at anytime.

Model. This component implements the logic of the application and exports an interface to access and manage the application's state. A model can be as simple as an integer with associated methods to increase, decrease and retrieve its value and representing a counter, or as complicated as a specific data structure with some related methods representing information about a document concurrently manipulated by a group of users. The model maintains a list of registered presentations and it is responsible for notifying presentations about changes in the state of the application's state, which keeps all presentations updated. The application framework does not impose any restriction on the implementation of the model, which can be built as a single component or as a collection of distributed components.

Presentation. The presentation transforms the application's state into a perceivable representation, such as a graphical or audible representation, a temperature or lighting variation, or in general, any external representation that affects the user environment and can be perceived by any of the human senses. The presentation generalizes the scope of the view component of the MVC, which was originally defined as a graphical representation rendered on a display.

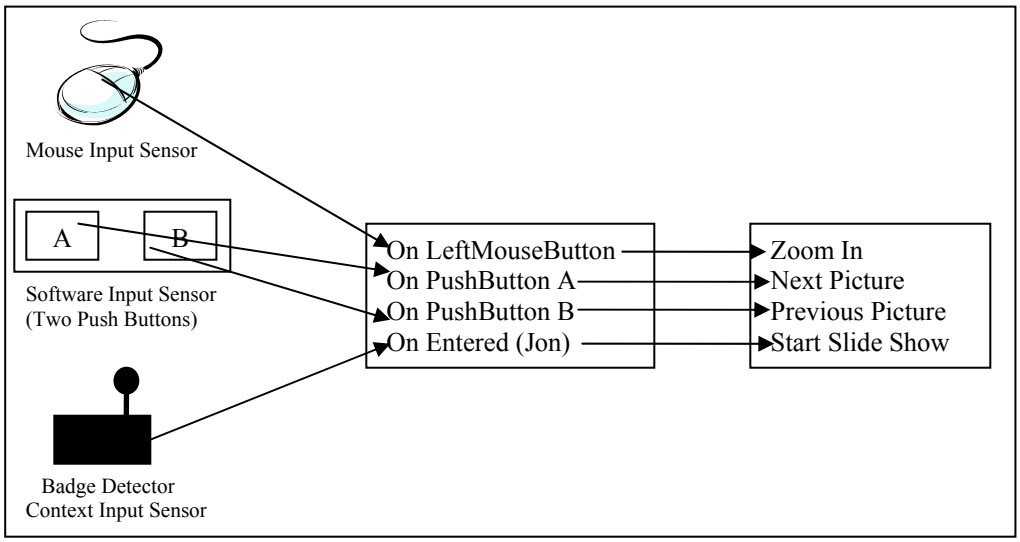

Figure 4. Example of a Controller mapping events from a Mouse Input Sensor and a Software Input Sensor (GUI with two push buttons) into method requests to the model

Controller. This component coordinates the interaction between input sensors and the application. It maps events generated by input sensors into requests to the application model that alter the state of the application. The model automatically notifies all attached presentations. The MVC uses the input sensor abstraction to refer to any hardware device capable of generating events of interest for the application. In most MVC applications these input sensors are the mouse and the keyboard. We use the term input sensor to refer to any entity (i.e., hardware and software) capable of generating events that can alter the application's state. Examples of hardware input sensors are mice, keyboards, active badges, i-buttons, and fingerprint detectors. Examples of software input sensors are GUIs containing widgets that can be associated to user defined events, and context input sensors, which are entities that process different context properties and synthesize specific context events. Modeling context as an input sensor has all the benefits described by Salber et. al. [12], and simplifies the development of applications that can easily react to changes in the context.

Controllers allow defining the actions to take when a particular event from an input sensor is received. These actions can be as simple as a translation from the event name to the name of a method implemented by the model, or as complicated as a script that uses the event name, its associated parameters, and other input variables (e.g., context properties) to decide what method(s) to call in the model. The application framework can leverage existing context systems such as the Context Toolkit as input sensors. Figure 4 illustrates an example of a controller translating the events received from three input sensors into method requests for the model.

According to the proposed application framework, it is possible to associate more than one controller with the same application. Depending on configurable properties (e.g., type of input sensor, user utilizing the input sensor, or context properties such as location) different controllers can be activated at different times, therefore changing the effect of input sensors on the application. For example, the software input sensor's push buttons depicted in Figure 4 could be mapped to different method requests for the model, depending on the user's location, or time of the day.

Coordinator. The coordinator encapsulates information about the application components' composition (i.e., application meta-level) and provides an interface to register and unregister presentations, controllers, and inputs sensors, and retrieve information about the application's components at run-time. The functionality provided by the coordinator offers fine grained control over the application internal composition rules. This behavior contrasts with traditional MVC applications that define the composition rules for the application components statically - what views to connect to the model and what controllers to use with the views.

Applications for ubiquitous computing environments are subject to constant changes including resources present in the users' environment, availability of the resources, and changes in the application's context (e.g., location, time, number of people, and current task). The functionality provided by the coordinator allows applications to alter their internal composition to better adapt to changes. For example, a user entering an active office containing several plasma displays may want to move the calendar application from his or her PDA to the active office. As a result, the application should reconfigure itself to use all plasma displays to present different views of the calendar simultaneously (e.g., monthly, daily, and weekly view), and should use a touch screen, a keyboard, and speech recognition simultaneously to accept data and commands from the user.

Specialization Mechanisms

Traditional operating systems provide mechanisms to abstract hardware details from application developers, so that applications built for a particular operating system can be used in machines with different hardware configurations. Our application framework seeks the same type of functionality, that is, allow developers to build applications that can be used in heterogeneous ubiquitous computing environments (active spaces). Users should be able to use the same applications in their active home, active car, or active office.

Ubiquitous computing environments pose two main challenges related to resources: heterogeneity and number existing resources. Traditional operating systems successfully address the issue of heterogeneity by providing software abstractions to represent the real hardware devices and mapping those abstractions to the specific details of the existing hardware. However resource cardinality is not normally a concern in traditional operating systems, which can assume certain hardware configurations. For example, most personal computer operating systems can safely assume the existence of peripherals such as one monitor, one keyboard, one mouse, one audio device, one video card, and some storage device. Unfortunately, this does not apply to ubiquitous computing environments. While an active meeting room can have several devices such as displays, keyboards, and mice, an active car may not have any display, keyboard, or mouse. However, it may offer additional resources (e.g., speakers, and microphone) that make it possible to use the application prior to dynamic adaptation of the application.

Applications built with our application framework are independent of a particular active space by using generic application descriptions that list the application requirements. These descriptions are used to create a specific application description that includes specific resources present in the active space that match the application requirements. The framework defines two types of application description files: the application generic description (AGD), and the application customized description (ACD).

The AGD (Figure 5) is an active space-independent application description that lists the components of an application and their requirements. The AGD uses name-value pairs to describe the requirements and components of an application and acts as a template from which concrete application configurations (i.e., ACDs) are generated. The description contains a list of application components consisting of one model, one coordinator, zero or more presentations, zero or more controllers, and zero or more input sensors. Every component entry includes a component name, an optional field with the parameters required, a field with the component cardinality (minimum and maximum number of instances of the component allowed), and a list of requirements for the component. These requirements include information such as for example, required operating system, and hardware platform. The specialization mechanism uses the requirements to query the active space infrastructure to obtain a list of matching entities. The syntax for the requirements is currently based on the CORBA's Trading Service query language [13]. The entry for the controller is optional, and in most cases is not used because presentations automatically register default controllers (this is the case of the example listed in Figure 5).

```
Model {
  ClassName  JukeboxModel
  Params      -f <files' location>
  Cardinality  1 1
  Requirements device=ExecutionNode
           and OS=Windows2000
}
Presentation {
  ClassName   MusicPlayer
  Cardinality   1 *
  Requirements device=ExecutionNode
           and type=AudioOutput
           and OS=Windows2000
}
Presentation {
  ClassName   ListViewer
  Cardinality   1 *
  Requirements device=ExecutionNode
           and OS=Windows2000
           or OS=WindowsCE }
```

```
InputSensor {
  ClassName   VCRInputSensor
  Cardinality  0 *
  Requirements device=ExecutionNode
           and device=Touchscreen
           and OS=Windows2000
           or OS=WindowsCE
}
Coordinator {
  ClassName  Coordinator
  Cardinality  1 1
  Requirements device=ExecutionNode
           and OS=Windows2000
}
```

Figure 5. Music Jukebox AGD.

The ACD is an application description that customizes an AGD to the resources of a specific active space. The ACD consists of information about what specific components to use, how many instances to create, where to instantiate the components, and what parameters to provide. ACDs are implemented as Lua scripts [14].

Figure 5 presents the AGD defined for an application called Music Jukebox, which provides functionality to organize and play a collection of music files using resources present in the ubiquitous computing environment. The model, for example, is implemented by a component named MusicJukeboxModel, requires a parameter with the location of the files (we use the Gaia File System to aggregate songs stored in different devices), has a cardinality of one (a Music Jukebox application has exactly one model), and requires an ExecutionNode device running Windows 2000. Gaia uses the term Execution Node to abstract any device capable of hosting the execution of Gaia components (e.g., model, presentation, controller, input sensor, and coordinator).

Modes of Operation

The specialization mechanism has two modes of operation: manual and template-based. In manual mode, users create an ACD starting from a specific AGD and a target active space. In template-based mode, the application framework specialization mechanism uses the rules defined by a template and the resources contained in a specific active space to generate ACDs automatically. The resulting ACD can be used as it is or it can be further refined by users. The template-based mode does not require user input, and therefore can be used to create default ACDs for new environments. For example, the *Maximize Template* generates ACDs that instantiate the presentations and input sensors in all compatible devices available in the active space, according to the maximum cardinality of each component, and instantiates the model, controller(s), and coordinator in any compatible device.

Figure 6 illustrates two ACDs manually generated from the AGD (Figure 5) using the graphical tool illustrated in Figure 7. The ACD on the left is customized for a prototype active meeting room, and the ACD on the right is customized for an active home environment.

```
Application = {
  Model = { {
  ClassName ="JukeboxModel",
    Hosts = {{ "amr1.as.edu",
                 "-f /owner/mp3" },}
  } },
  Presentation = {
    ClassName ="MusicPlayer",
    Hosts = { { "amr2.as.edu","" }}
    } },
  Presentation = { {
    Classname ="ListViewer",
    Hosts = {{ "plasma1.as.edu",""},
               { "plasma4.as.edu",""},
  } },
  InputSensor = { {
    Classname ="VCRInputSensor",
    Hosts = {{ "pda.as.edu",""},}
  } },
  Coordinator = { {
    ClassName ="Coordinator",
    Hosts = {{ "amr3.as.edu","" },}
  } },
}
```

```
Application = {
  Model = { {
  ClassName ="JukeboxModel",
    Hosts = {{ "livingroom.as.home" ,
    "/owner/mp3"},}
  } },
  Presentation = {
    ClassName ="MusicPlayer",
    Hosts = { { "livingroom.as.home","" },
          }
    } },
  Presentation = {
    ClassName ="ListViewer",
    Hosts = { { "livingroom.as.home","" },
          }
    } },
  InputSensor = { {
    Classname ="VCRInputSensor",
    Hosts = {{ "pda1.as.home"},}
  } },
  Coordinator = { {
    ClassName ="Coordinator",
    Hosts = {{ "bedroom.as.home","" },}
  } },
}
```

Figure 6. Music Jukebox ACD for a meeting room scenario (left) and Music Jukebox ACD for a home scenario (right).

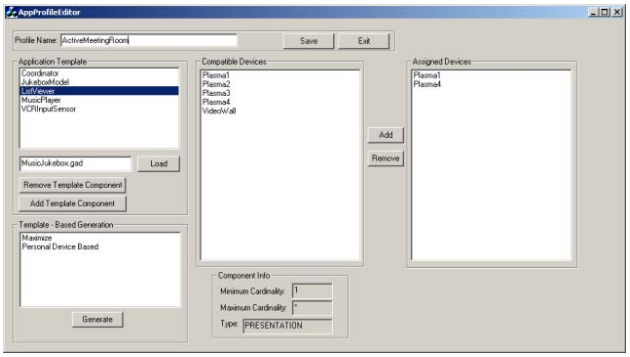

Figure 7. Application Specialization Tool

Application Management Policies

The application framework provides interfaces to inspect and modify the composition of the application components. These interfaces are essential to adapt the application to different scenarios. However, the type of application adaptation depends on different issues including the goal of the adaptation, and the nature of the changes. We use external policies (e.g., scripts, and services) that use the interfaces exported by the application framework to drive the adaptation of the application. These policies can be as simple as a script that checks a simple condition (e.g., temperature) to decide how to alter the applications, or as complicated as an AI algorithm that adapts the application based on the user behavior. We use policies for application instantiation, customization, reliability, and mobility, and we also plan to use policies for security in the future. We describe next the goal of each of the four policy types.

Application Instantiation. Applications based on the application framework are composed of a collection of distributed components. As a result, the application instantiation mechanism must take into account the

possibility of components crashing during the instantiation, and must define the actions to take in case of failures. We define two default policies for application instantiation: strict and best-effort. Both policies are implemented as scripts that parse an ACD and instantiate the application. The *strict* policy guarantees that the application will be instantiated only if all components of the application are successfully created and connected. The *best-effort* policy guarantees that the application will be started if the model, coordinator, controller, and at least one presentation and input sensor are successfully created and connected. This policy is useful in situations where the application has duplicated presentations and therefore, if some of the presentations crash it does not affect the usability of the application or in situations where the user can manually correct the problems attaching additional presentations or input sensors.

Context Customization. These policies modify the composition of applications according to changes in the context. For example, a user reading a confidential document on an active office display may define a policy that moves the presentation to his personal PDA if someone else enters the office. Different active spaces (e.g., active office, active home, and active mall) offer collections of specific policies to customize applications to such environments, and users have their own collections of policies according to their personal preferences.

Application Reliability. When the components of an application are distributed among a number of devices, the risk of failure increases (e.g., network errors, device failures, component errors). However the application reaction to failure depends on the type of failure, the nature of the application, and the user and active space preferences. For example, if the play list presentation of a music application crashes, the application may simply continue and update the state of the coordinator, or may decide to restart the component automatically. A real-time video application that detects that the response time of the presentation is not adequate (e.g., video output out-of-synch) may transform the video format to a less bandwidth demanding encoding, or may decide to terminate the application and notify the user. We currently use a default reliability policy that detects when an application component stops functioning and automatically detaches it from the application using the coordinator interface.

Application Mobility. Mobility policies allow users to define when to move an application, as well as what components of the application to move, under what conditions, and where to store the state of the application. Policies interact in most of the situations with the active space infrastructure, so they can learn about changes in the context, or variations in user preferences, and can therefore initiate the migration of the application.

Experimental Results: Music Jukebox application

We present in this section the Music Jukebox Application, an application based on our application framework that provides functionality for playing music files taking benefit of the resources contained in the active space where the application is instantiated. The application base-level provides functionality for managing a collection of music files distributed among different devices located in different active spaces, allows selecting, controlling, and playing a specific song in the user's current location, and exports information about the play list contents, as well as the currently selected song. The application provides also functionality to register, unregister, duplicate, and move presentations and input sensors dynamically, adapts to context changes, and uses mobility policies to follow the user to different active spaces.

Implementation Details

In this example we focus on our active meeting room (Figure 1), managed by Gaia OS. For each machine in the meeting room, the Gaia bootstrap protocol starts specific services according to the resources of the machine (e.g., infrared beacon service, audio service, and badge detector service) and a service called *ExecutionNode* that registers the device with Gaia OS and exports functionality to create, destroy, and upload Gaia components in the device. PDAs also execute the *ExecutionNode* service, which has been implemented using an embedded CORBA ORB [15].

Figure 5 illustrates the AGD for the music application, which consists of a model, two presentations (player and playlist viewer), one input sensor (VCR), and one coordinator. The MusicPlayer presentation automatically instantiates a default controller that maps the events from the VCR input sensor (i.e., start, stop, next, previous, volume up, and volume down) into method requests with the same name sent to the model. The MusicPlayer presentation reuses an existing application called Winamp [16] to play the audio.

We have successfully implemented all the functionality presented in this application, and we use the Music Application on a regular basis. The response time of the application is within an acceptable range from an interactive point of view. For example, start, stop, next, and stop require less than a second to execute and manipulating the meta-level (duplicating, moving, attaching and detaching presentations and input sensors) takes from 3-6 seconds depending on the request.

Instantiating the Application

We describe in this section how the active meeting room instantiates the Music Application. The user enters the meeting room and points the PDA device at the infrared beacon. The beacon transmits a handle to the active space, which the PDA uses to resolve the Gaia OS services and register with the space via the wireless Ethernet interface. The registration process introduces the PDA as a resource of the active space and sends information about the user, which the session manager service uses to instantiate the Music Jukebox application with a strict instantiation policy (i.e., the application is instantiated only if all components are successfully created and connected) and the ACD illustrated on the left side of Figure 6. The policy is implemented as a script that parses the ACD, instantiates the components in the specified devices, and finally registers the model, presentations, and input sensor with the coordinator, which assembles the application. The diagram depicted in Figure 8 illustrates the resulting application partitioning.

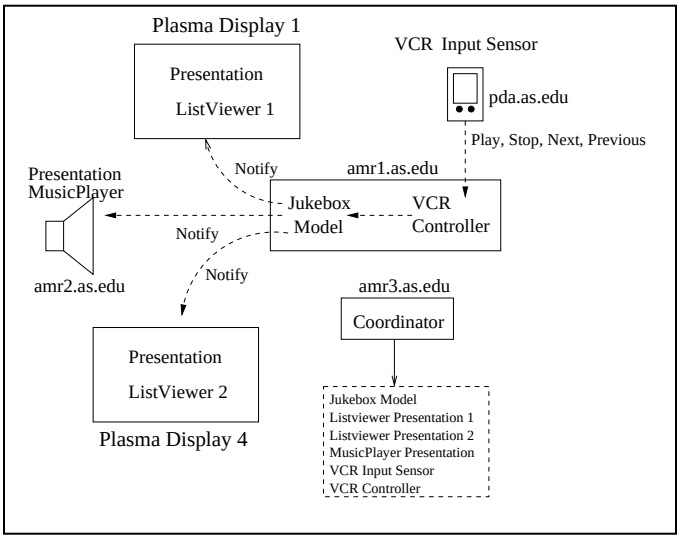

Figure 8. Music Jukebox Schematic. Application instantiated according to the Music

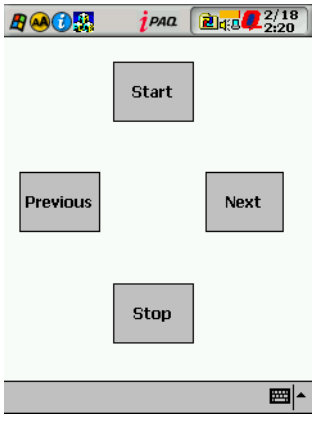

Figure 9. Pocket PC VCR Input Sensor

The user interacts with the VCR Input sensor running on the PDA (Figure 9) to control the music. Events generated by the VCR affect the audio output (MusicPlayer) as well as the name of the highlighted song (ListViewer). For example, when the user selects start, the VCR Input Sensor sends an event to the VCR Controller, which invokes the *startSong* on the Jukebox Model, which in turn sends a notification to all the attached presentations. The Music Player contacts the model to fetch the music file from its remote location, and interacts with Winamp to start playing the song. The ListViewer presentations are not affected by start and stop events.

The user can also interact with the coordinator of the application to alter the composition of the application. For example, the user can move, duplicate, detach and attach ListViewers to any display in the meeting room, including any available PDA. Users can also move the MusicPlayer to any Windows 2000 device equipped with audio output. The MusicPlayer automatically

resumes the audio at the point where it was stopped before it moved. We offer a graphical tool to manipulate the meta-level of any application running in the active space.

Related Work

The PIMA [2] and I-Crafter [3] projects propose a model for building platform independent applications. Developers define an abstract application that is automatically customized at run-time to particular devices. PIMA and I-Crafter generate applications for a single device, while we consider applications partitioned across devices. However, we can leverage the functionality provided by both approaches to dynamically generate application presentations customized to specific devices. The Pebbles [4] project is investigating partitioning user interfaces among a collection of devices. Pebbles is mostly concerned with issues related to GUIs, and the proposed infrastructure does not provide functionality for dynamically altering the partitioning layout. Our application model focuses on the application structure (logic, control, presentation, and meta-level management), application lifecycle, application adaptability and configurability, and provides reflective functionality that allows altering the application structure at run-time.

BEACH [5] is a component-based software infrastructure that provides support for constructing collaborative applications for active meeting rooms. BEACH applications are similar to the applications we propose in that they contemplate one user exploiting multiple devices at the same time, dynamic reconfigurations, integration of the physical space, interoperation among all resources contained in the space, and they rely on a software infrastructure to access resources contained in the space. However, the main differences between BEACH and our approach are that BEACH concentrates on collaborative applications while we consider both collaborative and single user applications, BEACH is customized for meeting room-like environments while our framework can be used in different scenarios. Finally our framework focuses on user-centrism and mobility, which allows us to move applications to different active spaces. BEACH does not directly address this issue because the target a single scenario. An interesting aspect of BEACH is the definition of reusable classes to simplify the construction of collaborative applications based on multimedia documents. This concept of class-libraries customized for particular environments is something we would like to incorporate in the future.

Graspable Interfaces [11] presents an evolutionary model for GUIs where physical objects are used to interact with applications. This approach distinguishes time-multiplexed input devices from space-multiplexed input devices. The most well known example of a time-multiplexed input device is the mouse. The same device is multiplexed over time to control different GUI widgets. On the other hand, the space-multiplexing model is based on the idea of associating specific physical objects to specific functional aspects of the application. The objects become dedicated functional manipulators. There is a one-to-one mapping from a particular object to a virtual function. An example of space-multiplexed device is an audio mixing console, where each slider is associated to a specific music channel. The ubiquitous application model we propose combines both concepts and defines the time-space-multiplexed model.

Conclusions and Future Work

This paper has presented our application framework for designing and building User-Centric, Resource-Aware, Context-Sensitive, Multi-Device, Mobile applications. These applications are bound to users instead of devices, can take benefit of resources present in the users' environment, can react to changes in the environment, and can be partitioned among different devices. The application framework defines a component (coordinator) that provides functionality to access and modify the composition of the application dynamically. The application framework implements a mechanism to define applications abstractly and manually or automatically specialize them to arbitrary environments. Finally, the application framework uses flexible policies to separate the basic application construction and modification functionality from particular strategies.

Our current experiments prove that the application framework simplifies the design and implementation process. Furthermore, the flexibility and dynamism of such applications has simplified the interaction with active spaces such as our prototype active meeting room. The application framework allows integrating existing components including COM objects (e.g., Power Point, Word and Excel). The framework extends the functionality of these components by allowing users to move the component across different devices, choose different input sensors, and even extend them for collaborative environments.

Although we have not completed the proposed customizable habitat vision yet, we believe that the applications framework presented in this paper is a valid solution to program existing computational rich spaces. We plan to use the proposed infrastructure in the Siebel Center (our new computer science department) to create collaborative applications, including classroom and meeting environments.

Acknowledgments
The authors would like to thank Christopher Hess for his valuable input and useful discussions about different aspects of the application framework.

References

[1] M. Roman and R. H. Campbell, "GAIA: Enabling Active Spaces," presented at 9th SIGOPS European Workshop, Kolding, Denmark, 2000.
[2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. B. Sussman, and D. Zukowski, "An Application Model for Pervasive Computing," presented at Mobile Computing and Networking, 2000.
[3] S. R. Ponekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments," presented at Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia, 2001.
[4] B. A. Myers, "Using Hand-Held Devices and PCs Together," in *Communications of the ACM*, vol. 44, 2001, pp. 34-41.
[5] P. Tandler, "Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices," presented at Ubicomp 2001: Ubiquitous Computing, Atlanta, Georgia, 2001.
[6] A. K. Dey, "Providing Architectural Support for Building Context-Aware Applications," in *PhD Thesis in Computer Science*. Atlanta: Georgia Institute of Technology, 2000, pp. 188.
[7] T. O'Reilly, M. Langley, and D. Flanagan, *X Toolkit Intrinsic Reference Manual for Version 11 of the Window System*: O'Reilly, 1995.
[8] G. D. Abowd and E. D. Mynatt, "Charting Past, Present, and Future Research in Ubiquitous Computing," presented at ACM Transactions on Computer-Human Interaction, 2000.
[9] G. E. Krasner and S. T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," ParcPlace Systems, Inc., Mountain View 1988.
[10] M. Roman, C. K. Hess, R. Cerqueira, R. H. Campbell, and K. Narhstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," University of Illinois at Urbana-Champaign, Urbana-Champaign UIUCDCS-R-2002-2265 UILU-ENG-2002-1709, February 2002.
[11] G. W. Fitzmaurice, "Graspable User Interfaces," in *PhD Thesis in Computer Science*. Toronto: University of Toronto, 1996.
[12] D. Salber, A. K. Dey, and G. D. Abowd, "The Context Toolkit: Aiding the Development of Context-Enabled Applications," presented at CHI'99, Pittsburgh, 1999.
[13] M. Henning and S. Vinosky, *Advanced CORBA Programming with C++*: Addison-Wesley, 1999.
[14] R. Ierusalimschy, L. Figuereido, and W. Celes, "Lua: An Extensible extension language," presented at Software: Practice and Experience, 1996.
[15] M. Roman, A. Singhai, D. Carvalho, C. Hess, and R. H. Campbell, "Integrating PDAs into Distributed Systems: 2K and PalmORB," presented at International Symposium on Handheld and Ubiquitous Computing (HUC'99), Karlsruhe, Germany, 1999.
[16] Nullsoft, "Winamp Player (http://www.winamp.com)."