# A Model for Ubiquitous Applications[1]

Manuel Román and Roy H. Campbell
{mroman1, rhc}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Av., Urbana, IL 61801

*Report No. UIUCDCS-R-2001-2223, UILU-ENG-2001-1730*

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

# A Model for Ubiquitous Applications

Abstract

*The paper concerns the construction of applications for active spaces that support mobile users in an unobtrusive ubiquitous computing environment that couples models and virtual representations to the physical world. Traditional applications are often statically bound to resources, context insensitive, directed toward sequential computation, and do not support the activities of a mobile user. We propose a model for mobile applications that support mobile users, devices, and services in which the interactive nature of information and physical space is represented. We build on existing technology, extending the notion of the Model-View-Controller pattern to accommodate the requirements of ubiquitous computing.*

**Key Words:** Ubiquitous Computing, application model, mobility, adaptability.

## 1. Introduction

We present in this paper a model for ubiquitous applications involving mobile users in an unobtrusive ubiquitous computing environment that couples models and virtual representations to the physical world. We will refer to the infrastructure that supports these applications as an Active Space. Gaia [1] is an example of such an infrastructure. We argue that traditional methods of building applications are inadequate to exploit the many new properties of ubiquitous computing. Instead, we propose a new model for ubiquitous applications that includes support from an extended Model-View-Controller (MVC). We describe the requirements for ubiquitous applications, our solution, and its implementation based on Gaia OS. We will show how various applications benefit from the model and compare our approach with that of others.

### *1.1 Motivation*

Consider a ubiquitous computing example in which a user presents slides in a seminar. A key part of a traditional application would be a slide show viewer that displays slides stored in a file on the standard output display attached to a computer. The application is controlled through the keyboard and mouse. However, the traditional application is inadequate to describe how a user might benefit from a ubiquitous computing environment. For example, the user may want to display the slide contents on projector 1. He may want to display the other slides on a plasma panel. Instead of a keyboard, the user may want to use gestures. Members of the audience with poor eyesight might read the slides with a personal networked braile reader. The MVC underlying the slide show viewer application depends on specific devices like the keyboard and display. The dependencies that are inherently programmed into the application support cannot accommodate the needs of computing in a ubiquitous environment.

### *1.2 Contents of Paper*

We present in this paper a model for ubiquitous applications (or Active Space based applications) that defines how to build applications that take full benefit of the Active Space infrastructure. This model takes into account issues such as application decomposition, mobility, dynamic adaptability, automatic data adaptation, and existence of diverse input and output mechanisms. This model is not customized for a particular type of application running on a specific Active Space. Instead is generic enough to allow building arbitrary applications, running in different Active Spaces. The model allows casual configuration of applications, which simplifies the task of dynamically adapting applications.

The paper is organized as follows: section 2 compares a traditional application with its ubiquitous counterpart. Section 3 lists the main properties of ubiquitous applications. Section 4 describes our proposed model for ubiquitous application. Related work is presented in section 5, and the paper concludes in section 6 and presents our future work in section 7.

## 2. Traditional vs. Ubiquitous Application

In this section, we examine the many issues of building applications and how a ubiquitous environment challenges existing techniques. To simplify the presentation, we base the discussion on an extended slideshow application that we have implemented in Gaia, using our application model. To start, we describe how a person uses an application such as Microsoft Power Point built using a traditional approach. Then, we present a new scenario using our ubiquitous application model and compare them.

In the first scenario, the speaker enters the room, walks to the keyboard of the computer attached to a projector, and starts the presentation. The slideshow application displays the slides using the output device physically attached to the computer, in this case the projector. The speaker controls the presentation through any of the input devices connected to the computer (e.g. mouse and keyboard). Should the speaker want to display a full-screen video during the slide presentation, using the same projector, a resource conflict occurs. To resolve the conflict, the speaker could restart the slideshow in another computer connected to a different output device while showing the video on the projector. Conflict resolution and resource allocation is not automatic. Finally, while presenting the slides, the speaker moves around the room. In order to remotely control the presentation, he can use a wireless mouse or keyboard. The input format is supported by hardware redirection.

This scenario implies application, input, and output interface mobility but this mobility requires manually altering the physical environment, restarting the presentation in another device, and attaching specialized input devices to the equipment. While the approach allows resource reassignment, resource allocation is not automated and the application is firmly constrained to the physical devices used.

We now compare the previous scenario with a new one where we replace the traditional slide show application with our ubiquitous slide show application. We assume that the conference room is an Active Space which supports the execution of ubiquitous applications. The speaker enters the room, but does not have to walk to the keyboard of the computer attached to the projector. Instead, he/she interacts with the Active Space (e.g. voice recognition or touch screen) to start the slide show application. The Active Space automatically searches for appropriate devices and presents a list to the speaker. The speaker selects the projector for the slides and his/her PDA to control the presentation. The Active Space instantiates the components required for the application in the selected devices (the view of the application in the projector, and the controller in the speaker's PDA) and starts the presentation. During the presentation, the speaker uses the projector to display a full-screen video. The Active Space detects the resource conflict and automatically reconfigures the slide show application. The application is adapted so it displays the slides in the PDAs of the people in the audience. The speaker also requests controlling the presentation using gestures while the video is displayed. When the video finishes, the Active Space automatically moves back the slides from the PDAs to the projector, and restores the controller in the speaker's PDA.

The model for ubiquitous applications provides support for casual configurations. The ability to move the output and the controller applies not only to the slide show application, but in general to any application based on the proposed model. Unlike traditional applications, ubiquitous applications are not constrained to a particular device and can be easily modified and partitioned at run-time. Consider the speaker from the previous example is late due to a traffic jam. He/she could start the presentation from his car, and once he/she gets to the conference room the Active Car would interact with the Active Conference Room to transfer the components of the application running in the car.

The application model we propose provides the functionality required to build applications for Active Spaces. The dependencies of these applications are not statically programmed but instead are externalized. Therefore, the services provided by the model for ubiquitous applications can modify and inspect these dependencies, in order to accommodate mobility and adaptability requests.

## 3. Ubiquitous Applications Issues

According to the example presented in section 2, it is clear that traditional applications do not work in the context of Ubiquitous Computing scenarios. These new scenarios require a new type of applications which must be more flexible and adaptable. We present in this section some relevant properties for ubiquitous applications.

### Application Ownership

Traditional applications are "owned" by devices. Users have to utilize these particular devices in order to be able to access the application functionality. Therefore, too much emphasis is placed in the devices, complicating in some cases user-application interaction. However, in the case of Ubiquitous Computing, applications cannot be permanently associated to a single device, or to a collection of well-defined devices.

Instead, applications are partitioned and mapped into the most appropriate resources according to specific properties, such as physical context (e.g. user location, light intensity, and number of people present). Application ownership is transferred from particular devices to users and spaces. As a simple example of a user owned application consider a music jukebox application. The application is associated to the user, and as the user context changes, the application automatically finds and uses the most appropriate resources to continue playing and controlling the music. For example, the user may initially be in his/her office and move later to his/her car. While in the office, the music application uses one pair of computer speakers to play the music, retrieves the music from a distributed storage that contains files located at the user's home and at the office, and uses a customized GUI that runs in the user's PDA to control the song list as well as the music volume. As the user moves into the car, the application releases the speakers from the office and automatically uses the speakers of the car. In the same way, the previous GUI running on the PDA and used as the controller is released and the application automatically uses the buttons located in the steering wheel of the car to control the music. The songs, though, are still retrieved from the user's house and office. In this scenario, the user perceives a single music application that follows him/her as he moves. This contrasts with traditional applications where the user would have to learn and use two different music applications, one at the office, and another one at the car. The emphasis is in the application itself, instead of the devices used to execute and control the application. As an example of a space application consider an airport and an application that detects passengers entering the airport. The application automatically gives passengers personalized instructions regarding the status of their flight, and a navigational tool to guide them in real time to the gate. In this example, it is difficult to conceive the application in the context of a single device. The application does indeed run "in the context of the airport" using the most appropriate resources at any particular time (these resources include airport resources and resources owned by the passengers, such as PDAs and phones).

### Application Adaptability

Application adaptability refers to two fundamental aspects of the application. On the one hand it deals with adapting the application architecture, and on the other hand it refers to application data adaptation. Application architecture adaptability is concerned with the structure of the application; that is, the configuration of the components that constitute the application (normally referred as the meta-level of the application). These components can be classified into three main groups according to the functionality they implement. Some components are responsible for manipulating the data of the application, other components implement functionality to manipulate such data, and finally, another group of components is responsible for presenting the data to users. Applications must be capable of adapting their architecture as a result of external or internal events. According to the example presented in section 2, the user originally uses his/her PDA to control the presentation. However, when the video is started, the application changes the controller to gesture recognition. Likewise, when the video starts, the application releases the projector and uses instead the PDAs owned by the people in the audience to display the slides. Regardless of the configuration used, the goal of the

application is still the same, but the way we interact and receive results from the application changes. This architectural adaptability may in some cases require data adaptation. Going back to the example in section 2, when the slides are sent to the PDAs, they have to be adapted to the specific properties of each PDA. Some devices have a color display and are capable of decoding GIF images, while other devices have a grayscale display and cannot decode GIFs. As a result, the slides are automatically transformed from their original format to the format required by the different PDAs.

### Application Mobility

Transferring application ownership from devices to users and spaces spans the physical limits of individual devices and makes it possible to associate a computation to these users and spaces. However, this functionality requires ubiquitous applications to be mobile, that is, capable of migrating the different pieces of the application to different devices. This mobility imposes some requirements on the application architecture. It requires applications to be composed of distributed components capable of seamlessly communicating both locally and remotely, and the application must be able to migrate some of the components, without affecting the global consistency of the application. Application mobility is not an issue on traditional applications, which normally are confined to a particular device. The only way to continue with the execution of an application in another device is by explicitly starting the application in the new device. Application mobility involves in some cases adaptation. This is the case of the application presented in example 2, when the slides are sent to the PDAs.

### Importance of Context

Traditional applications are in most of the cases disconnected from the context associated to the user. Therefore it is not possible for these applications to customize their behavior to the particular user's situation. This differs from Ubiquitous Computing scenarios, where context can play a major role [2]. In Ubiquitous Computing scenarios context strongly affects applications and therefore it must become part of the application model. In the example presented in section 2, when the user moves from the car to the conference room, the application benefits from the user's location information to automatically move the controller. However, more complex configurations are possible. Assume that the conference room described in section 2 has two projectors. The application could use contextual information about orientation of people in the audience (where are they looking at) to decide which projector to use.

### External Infrastructure Support

Users' activities take place in particular physical spaces (e.g. meetings are held in meeting rooms, cooking takes place in a kitchen, and giving a presentation normally happens in a conference room). Therefore, it is reasonable to assume the existence of a software infrastructure that manages the space resources and supports the execution of ubiquitous applications or tasks, as well as the existence of network connectivity that allows interaction among all devices and services present in the space. Projects such as

the Interactive Workspaces from Stanford [3], Microsoft's EasyLiving [4] also assume the existence of an external infrastructure.

Traditional applications (device owned) rely on the functionality provided by the device operating system. This operating system abstracts the complexity of the device and exports and manages all the available resources in a standard way. From the perspective of ubiquitous applications, a similar type of infrastructure is required to simplify the development and usability of such applications. We call such software infrastructure an operating system for physical spaces, and it is a metaoperating system that runs on top of existing operating systems (e.g. Windows, Linux, Unix, WindowsCE and PalmOS) and presents physical spaces as programmable entities or simply "active spaces", which hide all heterogeneity (software and hardware) and export a homogeneous programming interface for a set of well defined services (e.g. discovery, space repository, security, event management, and data manipulation). We believe that an infrastructure such as the one proposed in this section (OS for physical space) simplifies the task of developing applications that fully exploit the potential of Ubiquitous Computing.

The ubiquitous application example (slide show presenter) we present in section 2 shows the importance of an external infrastructure. When the speaker walks into the room, the discovery service running in the Active Space detects the user and authenticates him/her (therefore he/she is allowed to use the resources contained in the space). The user decides to start the application and automatically receives a list of possible devices. This list is created by the Active Space, which retrieves the information from a Repository Service. When the user selects the devices, the Active Space again uses a service to dynamically instantiate the required components, in the most appropriate devices. The infrastructure implements functionality which is required for most applications running in the space therefore simplifying the development of such applications. The model for ubiquitous applications we propose requires the existence of an external infrastructure.

## Application Architecture

According to the properties previously mentioned, we believe it is essential to reconsider not only the way we develop applications but also the way we use these applications. From the development point of view it is important to move away from monolithic device-centric applications. The internal architecture of applications must be modular and loosely coupled, in such a way that it is possible to migrate, replace and adapt different components as required by the execution environment, while maintaining application coherency and consistency. Components must also be able to effectively and seamlessly interact locally and remotely. A component-based architecture is a main requirement, not only from the point of view of adaptability and mobility, but also from the point of view of reusability. The same components can be used unmodified in different applications. From the usability point of view, ubiquitous applications pose a new challenge regarding the way users activate and interact with applications. Traditional device-centric applications are predefined. That is, applications already define what are the specific components that compose the application, and can assume a default input mechanism (keyboard, mouse and pen) as well as a default output mechanism (display). While it is

possible to use component-based technologies such as CORBA [5], COM [6] and JavaBeans [7] to build such applications, this composition is done at development level, that is, developers choose the components they require for the application and implement the application logics that define the interaction rules among components. Therefore, users have no control over the structure of the application.
Due to the nature of Ubiquitous Computing scenarios, it is not possible to create predefined applications that assume a set of well-known resources. Ubiquitous applications are dynamically assembled on demand whenever their functionality is required. The sheer amount of services and devices present in Ubiquitous Computing scenarios complicates the process of creating applications that can assume a specific set of resources. However, even if it is feasible to assume a specific set of resources, users must be allowed to customize applications according to their personal preferences or requirements.

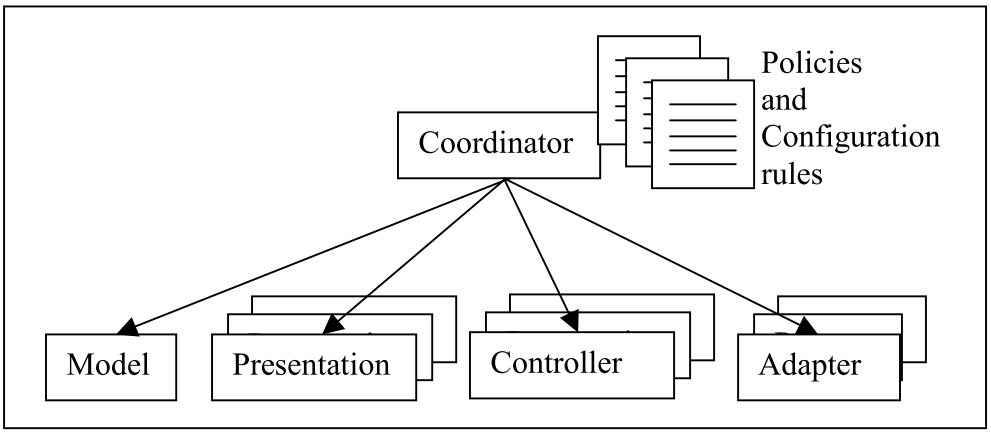


**Figure 1.** Model for Ubiquitous Applications

Consider again the example presented in section 2. To be able to move the different pieces of the application, the design of this application must be modular and loosely coupled so components can be migrated while preserving the integrity of the application. From the point of view of the user (the speaker), the application interaction paradigm is different from traditional applications. When the speaker enters the conference room, he has to choose the different pieces of the application, as well as their location. In the example, the slide viewer uses the projector and the controller runs in a PDA. The speaker has full control over the application architecture. The architecture of traditional applications is part of the application definition. Therefore users cannot take any decision about such structure.

## 4. A Model for Ubiquitous Applications

This section presents our proposed model for ubiquitous applications, which provides a standard mechanism for building applications for Ubiquitous Computing scenarios. This application model is based in the traditional Model – View – Controller paradigm, though

it adapts it (introduces to new components and defines new primitives) to Ubiquitous Computing scenarios.

### *4.1 The Traditional Model – View – Controller*

MVC [8] defines a modular scheme that models the behavior of any interactive application by clearly encapsulating (1) the model of the application domain (model), (2) the visualization of the model (view), and (3) the mechanisms to interact with the model (controller). This modular architecture simplifies the task of modifying and extending the application, as well as reusing specific components.

A model has one or more views attached, which are responsible for displaying the data in some particular way. The benefit of this separation between model and view is that the same model can be rendered in different ways. The model explicitly knows about the assigned views and is responsible for updating them whenever a change in the model's state is detected. The missing element in this scenario is the controller, which allows users to interact with the application model. A controller stores a model-view pair as well as a reference to an input sensor (e.g. mouse, keyboard and pen), which captures input events generated by users. The result of the input event (e.g. left mouse key pressed) depends on the associated view and the actions derived from the control mechanism are automatically sent to the view's associated model.

### *4.2 The Model – Presentation – Adapter – Controller - Coordinator*

The concepts defined by the traditional MVC are valid for any interactive application, regardless of the specific environment where applications run. An application has a model, externalizes such model so users can perceive it, and has some mechanisms to modify the state of the model. However, most of the existing implementations of MVC are customized for traditional application environments (device centric), and therefore it is difficult to reuse them in the context of Ubiquitous Computing. The Model-Presentation-Adapter-Controller-Coordinator (MPACC) is an application model that extends the MVC pattern to Ubiquitous Computing scenarios. This new model takes into account issues such as the non-existence of a single interaction device, contextual properties associated to the user and the space where the application runs, automatic model-view data type adaptation, mobility of the view, model and controller, and applications running on behalf of a user or a space, instead of in the context of a particular device. From the developers' point of view, the MPACC defines a model that standardizes the way in which applications for ubiquitous computing environments are built. From the user perspective, the ubiquitous application model defines a new way of using and customizing applications.

Traditional MVC presents the state of the model to users by means of views. Views are responsible for rendering the state in some visual form. In the ubiquitous application model, presenting the model's state to the user does not necessarily imply rendering it. The model can be externalized in any possible way that affects users' senses. Therefore in this new model, the user cannot only see the application's model, but also hear it, smell it, and touch it. One of the goals of this application model is to define a pattern that

standardizes the way in which applications for ubiquitous computing environments are designed, built and assembled.

The ubiquitous application model defines five elements: Model, Presentation, Adapter, Controller, and Coordinator. Figure 1 illustrates a schematic diagram of the application model.

An application is composed of a coordinator, a model, one or more presentations, one or more controllers, and optionally one or more adapters. The coordinator is responsible for managing the other four elements according to specific policies. The model for ubiquitous applications does not impose any restriction on the implementation of the components (e.g. they can be implemented as a collection of distributed sub-components or as a single component), this is orthogonal to the application model itself.

### *4.2.1 Model*

The Model is the implementation of the application's central structure, and consists of data and a programmatic interface to manipulate the data. 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 model that stores, processes and manipulates the vital signs of a patient in a surgery room, who is part of a medical application. Whenever the state of the model changes, it automatically notifies all attached presentations. The model has one or more presentations that externalize its state, an optional number of adapters that transform the state format to the presentation input format, and zero or more controllers that modify the state of the model.

The methods implemented by the model are *attachPresentation*, *detachPresentation,* and *change*. The first two methods are used to attach a new presentation and detach an existing presentation. The third method is called to notify all attached presentations that the state of the model has change. This method invokes the *update* method on all attached presentations.

### *4.2.2 Presentation*

Presentations are software components that transform an application model into an output format that can be perceived by human senses: touch, smell, sight, and hearing. This generalization contrasts with the traditional MVC implementations where the model is normally externalized through a view. Presentations have one associated model, none or one adapter and zero or more controllers. The association with the controller is used for scheduling reasons.

Presentations implement three methods, namely *setModel*, *getModel*, and *update.* The coordinator of the application calls setModel when the presentation is attached to a particular model. GetModel is used to obtain the currently associated model, and finally, update is called by the model to indicate that its state changed.

### *4.2.3 Adapter*

Models are not statically attached to particular presentations. Models are responsible for managing a specific type of data, and export well defined mechanisms to manipulate such data. Presentations, on the other hand, are responsible for interpreting such data and externalizing it. However, in some situations, data exported by a particular model cannot be directly manipulated by existing presentations. Therefore, the model's data has to be adapted to the data requirements of specific presentations. Adapters are the components responsible for this task. The decoupling of models and presentations makes it impossible for specific models to implement the adaptation mechanism. Models do not know a priori the type of presentations they will have attached. For this reason, adapters are instantiated automatically by the system (external infrastructure) when a presentation is attached to a model and the data types are incompatible. Adapters are associated to a single model, but they can be used by different presentations.

### *4.2.4 Controller*

The Controller encapsulates the mechanisms to modify the state of the model. However, unlike the standard MVC controller, the controller defined by the ubiquitous application model not only coordinates input devices, but in general any source of context.

Controllers provide the interface between the application model and any external data source (e.g. physical and digital context data) that can affect the application. Upon receipt of external data, the controller modifies the model accordingly. These changes are automatically propagated to all presentations associated to the model. Since it is possible for a user to have more than one device associated, it is also possible to have controllers running in different devices and therefore active at the same time. This contrasts the MVC where only one view/controller pair can be active at the same time. The ability of having more than one controller active at the same time allows space-multiplexed configurations [9]. This type of configuration allows having more than one controller, each one them associated to a different functional aspect of the application (e.g. a wheel to control the sound level, a switch to go to the next song and a touch screen to edit the play list of a music application). However, the ubiquitous application model introduces the concept of time-space-multiplexed configurations. Spatially multiplexed controllers can be time multiplexed according to different applications running at different times (e.g. the same wheel, switch and touch screen can be used to control a video projection application).

Controllers map input events into messages meaningful for the model. While it is possible to programmatically define these mappings (as in MVC), the ubiquitous application model allows users to configure the mappings at run time. These dynamic controllers allow a continuous customization of the behavior of the application according

to the user context and requirements. We use a scripting language called Lua [10] to rapidly define the actions to take upon receipt of specific events.

Controllers implement two methods: *setModel* and *getModel.* The application coordinator calls the first method when the controller is attached to a model. The second method is called to obtain the current model associated to the controller.

### *4.2.5 Coordinator*

The coordinator is responsible for managing the metalevel of the application (i.e. non application domain issues such as application architecture and policies regarding such structure). The coordinator stores references to the components that compose the application, as well as policies regarding adaptation, customization and mobility of the application. The coordinator can have zero or more controllers associated. These controllers notify coordinators about specific events, and as a result, the coordinator can modify the application. We call this controllers *meta-controller* since they affect the meta-level of the application. For example, a music application can have an associated policy specifying that whenever a user leaves the room, the controller and presentation of the application must be moved to the user's PDA. When the controller notifies the coordinator about the user leaving a room, the coordinator automatically detaches the music presentation and the controller from the room (e.g. speakers, and GUI running on a plasma display) and attaches a new music presentation and a new controller, which run in the user's PDA.

The controller implements fifteen methods: *setModel, getModel, registerPresentation, unregisterPresentation, registerController, unregisterController, registerMetaController, unregisterMetaController, getPresentations, getControllers, getMetaControllers, addPolicy, removePolicy, editPolicy and getPolicies.* SetModel assigns a specific model to the application, while getModel allows retrieving such model. The next four methods are used to register and unregister presentations and controllers with the application. When registering a presentation, the coordinator automatically invokes the *attachPresentation* on the model. The next two methods, *registerMetaController and unregisterMetaController,* attach and detach a controller to/from the coordinator (meta-controller). The next three methods list the references of the presentations, controllers and meta-controllers registered with the application. Finally, the last four methods allow manipulating the policies associated to the application.

## 5. Related Work

*Graspable Interfaces* [9] presents an evolutionary model for GUIs where physical objects are used to interact with applications, therefore making the access to the digital world more natural. *Tangible User Interfaces* (TUI) [11] are a generalization of graspable interfaces. This new approach presents a model that combines digital and physical entities (bits and atoms). The main difference with graspable interfaces is that in the case of TUIs, physical objects can be both the input and output of the application. In general, TUIs propose moving the GUI from the desktop's monitor to the physical space inhabited by the user. Therefore, the world becomes the interface. TUI defines a new graphical user interface model called MCRpd (Model-Controller-Representation physical, digital)

Tangible bits [12] are based on the traditional MVC [8]. This model reuses the model and controller components from MVC and replaces the view component by the physical and digital representation. Audio and video are examples of digital representations, while physical entities such as “bricks” [13]. This approach is solely based on using physical objects as controllers (physical objects are both representations and controllers).

Cooperstock et al [14] describe the lack of flexibility presented by existing GUIs to interact with a myriad of devices. It is hard for users to fully manipulate devices according to their needs. They introduce the concept of *Reactive Environment,* which is normally associated to a space responsible for the low-level management of all the contained devices. They present as an example a Teleconferencing application, however, this application is customized for the particular environment and for the specific devices contained in the room.

The Stick-e Note Architecture [15] promotes the use of context as a fundamental application data source. This architecture proposes extending user-interfaces beyond the computer display by attaching data to physical objects located in the environment of the user. Devices can then be used retrieve the data whenever they get close to particular objects. The Stick-e Note Architecture is built as an MVC application, and it also introduces some changes. The controller is used to route trigger events to the right views, and the source of these controllers is context data. The views implement an interface and define the actions they perform when they receive a trigger event.

Brewster et al [16] describe how to use non-speech sounds to convey information to the user. This is an example of an application that does not rely on a visual representation to present data to users.

Context has already been identified as one key property in mobile applications [17] [2] [18] [19] [20]. The behavior of applications and especially the output of the application can be greatly affected. Context must become a standard component in every application development, and for this reason, a new application model is required [21] that includes such context as standard orthogonal component.

There has been some previous work on formal descriptions of the Model – View – Controller [22], as well as frameworks based on a similar model and intended to provide a systematic design method [23]. These approaches provide formal tools to define the behavior of interactive applications. Ubiquitous applications are built upon an extended MVC design; therefore, all previous formal tools can be applied to abstract and formally describe their behavior.

## 6. Conclusions

This paper presents a model for ubiquitous applications. This new application model reuses the fundamental concepts introduced by the traditional Model-View-Controller and adapts them to the scenarios defined by Ubiquitous Computing. The model we propose exports an interface that allows casual configuration of arbitrary applications running in Active Spaces.

As defined by Abowd and Mynatt [24] applications running in the context of Ubiquitous Computing are better defined as tasks. These tasks are normally associated to users and do not have a clear time frame. They are started but do not have a clear end. The ubiquitous application model presented in this paper provides the tools to build such applications, by taking benefit of issues such as mobility and adaptability.

Finally, the model for ubiquitous applications not only proposes a new way of creating applications, but also a new way of using such applications. These applications are dynamically assembled, can change their structure over the time to adapt to different situations, and are not confined to specific devices.

## 7. Future Work

At the time of writing this paper, the ubiquitous application model is still in its initial stage. We have already implemented the application model framework in the context of Gaia OS. We have also built some applications based on the proposed model (e.g. slide presentation application and music application), and have implemented a tool to manipulate applications running in Active Spaces. Our future work includes the study of each of the different issues proposed in section 3, and the implementation of applications to validate our results. There is also still work to do regarding the format of application templates, application reconfiguration rules, services required for creating, assembling, modifying and deleting applications, and the use of scripting support to easily customize applications.

**8. Acknowledgements**

The authors want to thank Chris K. Hess, Renato Cerqueira and the Software Research Group for their valuable comments.

## References

1. Manuel Roman, Christopher K. Hess, Anand Ranganat, Pradeep Madhavar, Bhaskar Bortaku, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell and Dennis M. Mickunas. *Gaia OS: An Infrastructure for Active Spaces. -SUBMITED TO UBICOMP'01-*. In Proceedings of the *Ubicomp'01*, Atlanta, 2001
2. Anind K. Dey and Gregory D. Abowd. *Towards a Better Understanding of Context and Context-Awareness*. In Proceedings of the *CHI Workshop on The What, Who, Where, When, and How of Context-Awareness*, The Hague, Netherlands, 2000

3. Armando Fox, Brad Johanson, Pat Hanrahan and Terry Winograd. *Integrating Information Appliances into an Interactive Workspace.* IEEE Computer Graphics & Applications, 2000. 20(3).
4. Steve Shafer, John Krumm, Barry Brumitt, Brian Meyers, Mary Czerwinski and Daniel Robbins, ***The new EasyLiving Project at Microsoft Research***. 1999, Microsoft Rsearch.
5. Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Professional Computing Series, ed. Addison-Wesley 1120. Addison-Wesley. 1999.
6. Don Box. *Essential COM*, ed. Addison-Wesley 440. Addison-Wesley. 1998.
7. Richard Monson-Haefel. *Enterprise Javabeans*. 2nd Edition ed, ed. O'Reilly 472. O'Reilly. 2000.
8. Glenn E. Krasner and Stephen T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. 1988, ParcPlace Systems, Inc.: Mountain View.
9. George W. Fitzmaurice, *Graspable User Interfaces*, In *Computer Science*. 1996, University of Toronto: Toronto.
10. Roberto Ierusalimchy, Luiz Figueiredo and Waldemar Celes. *Reference Manual of the Programming Language Lua version 4.0*. Rio de Janeiro, Brazil. 1999.
11. Hiroshi Ishii and Brygg Ullmer. *Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms*. In Proceedings of the *CHI*, Atlanta, 1997
12. B. Ullmer and H. Ishii. *Emerging frameworks for tangible user interfaces*. IBM - Systems Journal, 2000. 39(3&4).
13. George W. Fitzmaurice, Hiroshi Ishii and William Buxton. *Bricks: Laying the Foundations for Graspable User Interfaces*. In Proceedings of the *CHI*, 1995
14. Jeremy R. Cooperstock, Sidney S. Fels, William Buxton and Kenneth C. Smith. *Reactive Environments: Throwing Away Your Keyboard and Mouse.* Communications of the Association of Computing Machinery (CACM), 1997. 40(9): p. 65-73.
15. Jason Pascoe. *The Stick-e Note Architecture: Extending the Interface Beyond the User*. In Proceedings of the *International Conference on Intelligent User Interfaces*, Orlando, Florida, 1997
16. Stephen Brewster, Grégoy Leplatre and Murray Crease. *Using Non-Speech Sounds in Mobile Computing Devices*. In Proceedings of the *First Workshop on Human Computer Interaction with Mobile Devices*, 1998
17. Anind K. Dey and Gregory D. Abowd. *The Context Toolkit: Aiding the Development of Context-Aware Applications*. In Proceedings of the *Workshop on Software Engineering for Wearable and Pervasive (SEWPC)*, Limerick, Ireland, 2000
18. Anind K. Dey, Gregory D. Abowd and Daniel Salber. *A Context-Based Infrastructure for Smart Environments*. In Proceedings of the *First International Workshop on Managing Interactions in Smart Environments (MANSE'99)*, Dublin, Ireland, 1999
19. Hans-W. Gellersen, Albrecht Schmidt and Michael Beigl. *Adding Some Smartness to Devices and Everyday Things*. In Proceedings of the *Third IEEE*

*Workshop on Mobile Computing Systems and Applications*, Monterey, California, 2000

20. A. Schmidt, K.A. Aidoo, A. Takaluoma, U. Tuomela, K. Van Laerhoven and W. Van de Velde. *Advanced Interaction in Context*. In Proceedings of the *Handheld and Ubiquitous Computing*, 1999
21. Tom Rodden, Keith Cheverst, Nigel Davies and Alan Dix. *Exploting context in HCI design for Mobile Systems*. In Proceedings of the *First Workshop on Human Computer Interaction with Mobile Devices*, 1998
22. Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*. In Proceedings of the *ECCOP/OOPSLA'90*, 1990
23. D. D. Cowan and C.J.P. Lucena. *Abstract Data Views: An Interface Specification.* IEEE Transactions on Software Engineering, 1995. 21(3): p. 229-243.
24. Gregory D. Abowd and Elizabeth D. Mynatt. *Charting Past, Present, and Future Research in Ubiquitous Computing.* ACM Transactions on Computer-Human Interaction, 2000. 7(1): p. 29-58.