# Dynamic Application Composition: Customizing the Behavior of an Active Space

Manuel Román, Brian Ziebart, and Roy H. Campbell
*Computer Science Department*
*University of Illinois at Urbana-Champaign*
*{mroman1, bziebart, rhc}@uiuc.edu*

## Abstract

*The proliferation of wireless networks, hand-held PCs, touch panels, large flat displays, sensors, and embedded devices is transforming traditional habitats and living spaces into ubiquitous computing environments, or active spaces. We envision a middleware software infrastructure that abstracts the heterogeneity of these environments and transforms them into programmable environments. This middleware infrastructure provides support to manage the resources contained in an active space (low-level functionality), support to develop applications (application-level functionality), and support to define interaction rules among applications (active space-level functionality).*

*In this paper, we present a mechanism called "application bridge" that implements active space-level functionality. Application bridges provide a simple, yet effective, mechanism to define dynamic application composition interaction rules that confer the active space a specific behavior based on a number of parameters, including context, application status, and user actions.*

## 1. 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 [1-4] should adapt the habitat to the user preferences, 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 interact with flexible mobile applications, define the function of the habitat, and customize its behavior according to different properties (e.g., personal preferences and current context). An active space is an integrated programmable environment that contains heterogeneous network connected devices, services, and applications coordinated by a context-aware distributed software infrastructure, and populated by a number of people performing different activities.

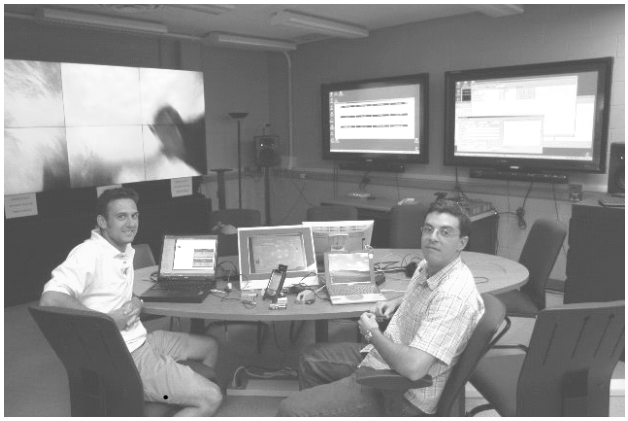

**Figure 1. Active Space Prototype**

Active spaces host the execution of different applications. We define an application as a collection of services that cooperate to achieve a common goal (e.g., play audio, control the lights, and present a slide show). For example, an active meeting room has applications to control the lights and the audio, present information in a ticker tape, control a slideshow, and track the number, identity and position of the people present in the room. According to our experience with a prototype active meeting room (Figure 1), a useful property of active spaces is the ability to orchestrate a number of individual applications to confer the active space a specific behavior. We identify three functional levels we consider essential to abstract a physical space and the resources it contains as a single homogeneous programmable environment:

- Low-level, which provides basic functionality including component management and resource discovery and is comparable to the functionality provided by traditional operating systems.

- Application-level, providing frameworks and tools to build applications.
- Active space behavior-level, which includes mechanisms to orchestrate the interaction among applications (dynamic application composition) and therefore provides functionality to program the behavior of the active space.

Existing research projects [5] [6] [7] [8] address the low-level and application-level functional issues but do not provide explicit support for active space behavior definition. We present in this paper an infrastructure to program the behavior of active spaces. The infrastructure simplifies the creation of customizable and dynamically-adaptable inter-application interaction rules that define how changes in an application affect other applications. We currently use the infrastructure to define interaction rules among six applications (i.e., speech engine, slide show manager, light controller, audio player, ticker tape, and location) running in our active space prototype. The results are encouraging and we have experienced a qualitative improvement in the global usability of the active space. Furthermore, it is possible now to perceive the active space as an interactive environment with a well-defined behavior instead of an execution environment consisting of disconnected applications.

The rest of the paper is organized as follows: section 2 describes the three functional levels of an active space, including low-level (section 2.1), application-level (section 2.2), and behavior-level (section 2.3); section 3 presents a detailed example of a ticker tape and a location application that use the bridging mechanism to interact; section 4 describes how to develop and use bridges; section 5 presents related work and we conclude the paper and describe our future work in section 6.

## 2. Active Space Functionality Levels

We have developed a meta-operating system called Gaia OS [9] to manage active spaces. Gaia is a distributed middleware infrastructure we refer to as a meta-operating system[10] that coordinates software entities and heterogeneous networked devices contained in a physical space. Gaia exports services to query and utilize existing resources, to access and use current context, and provides a framework to develop active space-aware applications. Figure 2 illustrates the architecture of Gaia OS.

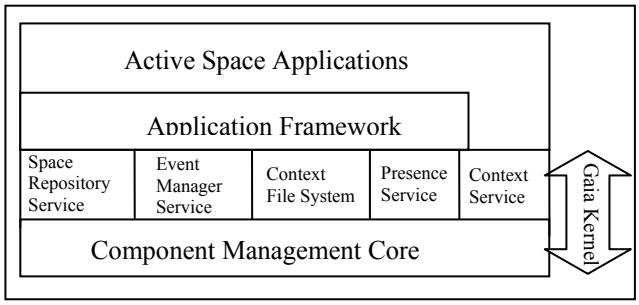


**Figure 2. Gaia OS Architecture.**

### 2.1 Active Space Low-Level Functionality

The Gaia OS Kernel provides services for location, context, events, and repositories with information about the active space. It is built as a distributed object system that extends the notion of an execution environment associated to devices to the space level. The kernel also provides functionality to manage remote components (e.g., create, destroy, load, unload, and transfer). Gaia OS abstracts the physical space and the resources it contains as a programmable execution environment.

The Gaia OS Kernel implements the active space low-level functionality and it is comparable to the functionality provided by traditional operating systems (e.g., process management, file system, and inter-process communication).

### 2.2 Active Space Application-Level Functionality

Gaia applications use a set of component building blocks, organized as the Gaia Application Framework [11] to support applications that execute within an active space. The framework provides mobility, adaptation, context-awareness, and dynamic binding. The functionality permits commercial off the shelf, as well as new applications, to run in the active space. The application framework models applications as a collection of distributed components, and reuses some concepts from the Model-View-Controller. 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 portable applications, and provides functionality to manage the application lifecycle (i.e. instantiation, adaptation, suspension and resumption, fault-tolerance, termination, and mobility).

The application framework infrastructure is composed of five components (Figure 3): model, presentation, controller, input sensor, and coordinator.

The model, presentation, controller, and input sensor are the application base-level building blocks and are strictly related to the application domain functionality.

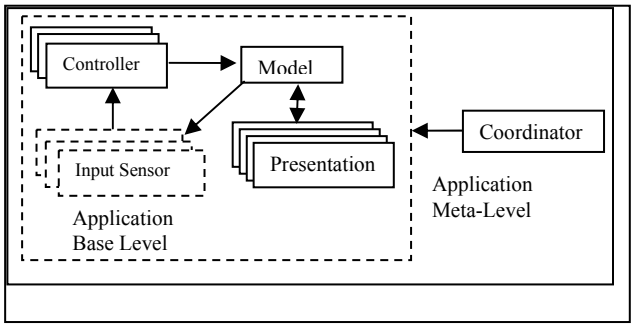


**Figure 3. Gaia Application**

The model implements the logic of the application and exports an interface to access and manage the application's state. The model maintains a list of registered listeners and it is responsible for notifying them about changes in the application's state, therefore keeping them synchronized.

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. Presentations can be dynamically attached and detached to and from the model, and are registered as listeners. Presentations are responsible for contacting the model and retrieving the new state upon receiving a notification.

The input sensor is the component responsible for changing the state of the application. Input sensors can be interactive (e.g., GUI and speech-recognition) or non-interactive (e.g., context synthesizers), and they interoperate with the model's interface to alter the state of the application. Input sensors receive notifications from the model so they can synchronize their internal state with the rest of the application.

The controller mediates the interaction between the input sensor and the model. It translates requests from the input sensor into method calls that match the model's interface, therefore maximizing input sensor reusability. The same input sensor can be used with different applications by changing the mappings stored in the controller dynamically (Figure 4).

The coordinator encapsulates information about the application components' composition (i.e., application meta-level), provides an interface to register and unregister presentations and input sensors, and allows manipulating the bindings of the application components. The coordinator provides functionality to retrieve run-time information about the application's components composition. The functionality provided by the coordinator offers fine grained control over the composition of the application base-level components.

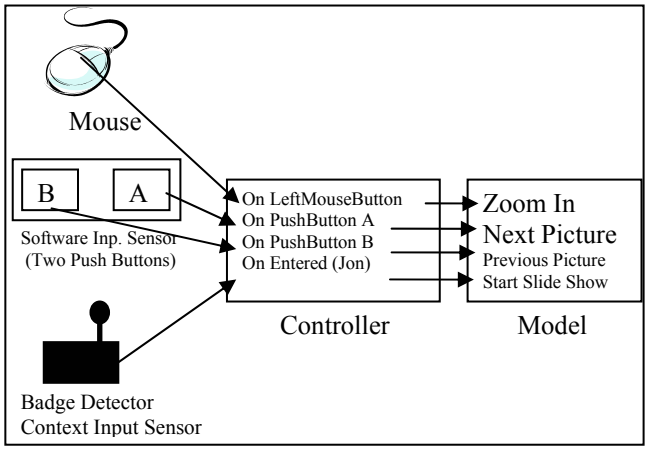


**Figure 4. Example of a Controller mapping events from a Mouse Input Sensor, a Context Input Sensor, and a Software Input Sensor into method requests to the model.**

## 2.3 Active Space Behavior-Level Functionality

Applications built using the application-level functionality are disconnected execution units. The Gaia OS application framework implements an additional mechanism called *application bridge* to support the active space behavior-level functionality. This functionality allows defining rules that specify how changes in an application affect the execution of other applications and therefore make it possible to program the behavior of the active space (i.e. coordinate all applications hosted by the active space).

The active space behavior-level functionality does not require any changes in the applications involved in the interaction, it is independent of the functionality implemented by the connected applications, and allows defining and modifying the interaction rules at run-time.

The application bridge (Figure 5) is an input sensor that listens for notifications from the source application and introduces changes in the target application by invoking methods on the model via the controller.

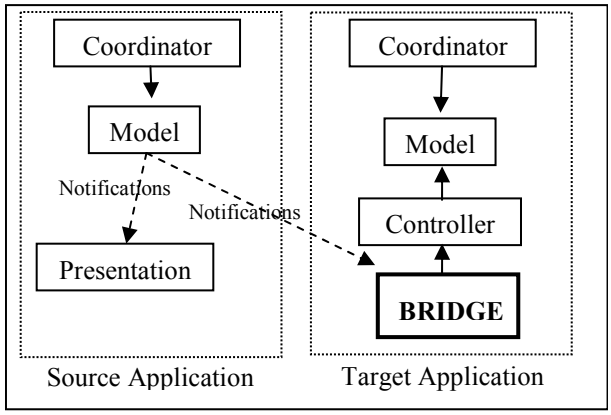


**Figure 5. Application Bridge (in order to simplify the diagram we only show the components involved in the interaction).**

The bridge implements functionality to execute user-defined rules that affect the state of the target application's model when it receives a notification from the source application. The mechanism to trigger the execution of the user-defined rules is common to all bridges while the rules defining what actions to take are bridge-dependent and are implemented as scripts. The script for the bridge receives a reference to the source application's model, a reference to the target application's controller, and the source application notification's hint (notification sent by the source application's model to inform about changes in its state). Developers write a script (current implementation is based on LuaORB[12]) using these parameters to define the interaction rules. The bridge executes the script each time it receives a notification from the source application model. Figure 6 illustrates the interface of the script.

```
function bridgeRules (
      targetController, sourceModel, sourceEvent )
            <Interaction Rules>
end
```

**Figure 6.** Application Bridge Script Interface.

## 3. Using a Ticker Tape to Display People Location

In this section, we describe a ticker tape application that uses several synchronized displays to present information, and a location application that provides people location information (room granularity). Next, we explain how we use the ticker tape application to display people location information using the active space behavior-level functionality.

### 3.1 Ticker Tape Application

This application provides support for displaying scrolling items sequentially across multiple display devices (Figure 7). The ticker tape serves as an input/output interaction mechanism within an active space. Unlike traditional stock quoting ticker tapes, our ticker tape displays multimedia items, including graphics, and allows assigning specific actions to the scrolling items.

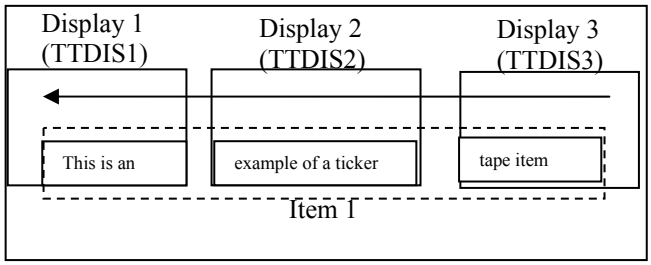


**Figure 7. Ticker Tape Item synchronized across three displays.**

Items displayed in the ticker tape can be selected, and they trigger user defined actions, including launching additional applications, or modifying the state of existing applications.

One main characteristic of the ticker tape is the synchronized and dynamic utilization of multiple display devices. Applications in an active space are not confined to one display device; therefore, a ticker tape item (e.g., text and pictures) displayed in an active space can be rendered on multiple devices. When a ticker tape item reaches the edge of one display, it is immediately displayed in the next display. In addition, components in an active space are often mobile, so the ticker tape must be able to respond to devices entering, exiting, and changing location within the active space by attaching, detaching, and re-ordering ticker tape items.

The Ticker Tape is composed of four components: Model, Ticker Tape Display Input Sensor (TTDIS), and Coordinator. The Ticker Tape implements the first two components and reuses the default Coordinator implementation provided by the application framework.

According to the application framework description discussed in section 2.2, the ticker tape application component that displays the scrolling items (TTDIS) should be a presentation (it displays the scrolling items). However, the ticker tape application allows assigning actions to the items, and therefore we model the TTDIS as an input sensor (input sensors receive notifications and therefore can implement presentation functionality).

The *Ticker Tape Model* orchestrates the synchronized handling of scrolling items across the different displays used by the application. The model associates an index to each scrolling item, and stores an ordered list of TTDIS ids (based on the scrolling item

display order), so it can dispatch notifications to the appropriate TTDIS when an item needs to be displayed. For example, based on the example depicted in Figure 7, the model stores an id for the scrolling item (Item 1), and an ordered list of TTDIS in the following order: TTDIS 3, TTDIS 2, and TTDIS 1 (the item scrolls from right to left). The model also implements functionality for adding, updating, and removing scrolling items. A scrolling item is stored in the model as a set of attributes, including size, color, font and content of text, the path location and size of pictures, and other attributes to determine how items are rendered and displayed by the display components.

The *Ticker Tape Display Input Sensor* (TTDIS) is responsible for displaying scrolling items in a display when the model sends the appropriate notification, and notifying the model when its scrolling item reaches the edge of the display so that the next TTDIS can be notified to display the item. In addition, the TTDIS is responsible for detecting and notifying the model when users select a certain scrolling item so that the model can execute any functionality associated with that item. Upon receiving a notification from the model to display a scrolling item, a TTDIS checks if the notification is intended for it. If so, it requests the set of attributes associated with the item from the model, and renders and displays the scrolling item.

Figure 9 illustrates all the ticker tape application components and explains all the steps required to scroll items.

## 3.2 Location Application

The location application provides functionality to track people inside our computer science building. The application relies on sensor data provided by the active space low-level functionality (Gaia Kernel) to detect the position of the users. The current implementation of the Gaia location application provides information at room granularity. That is, we can detect whether or not a user is present in a room, but not where in the room the user is located.

The location application implements three components, Location Model, Location Presentation, and Location Input Sensor, and reuses the default coordinator (Figure 8).

The *Location Model* provides functionality to store and update information about users and their locations and provides an interface to query about user location. The model stores information about the user name, the name of the space where he or she is located, and the date and time the user entered and left the space.

The *Location Presentation* is a graphical presentation that displays information about user location. Users can select a user name and get updated information about his or her position, or select a space and learn about the people located in it.

The *Location Input Sensor* leverages the low-level functionality (Gaia OS Kernel) to learn about users entering and leaving the space. When a user enters or leaves, the location input sensor updates the model's states via the controller. There is one instance of the input sensor for each active space.

Figure 8 illustrates the composition of the location application running in our building. We define three active spaces: domain, 2401, and 3231. These three active spaces are hierarchically organized as a tree, with the domain at the root and 2401 and 3231 as leaves.s The coordinator, model, and controller of the application run in the domain active space, and 2401 and 3231 host the execution of the location presentation and location input sensor. When a person enters 2401 or 3231, the input sensor sends a notification to the model running in the domain via the controller (steps A and B in Figure 8), which notifies the presentations (steps C and D). Tracking people in additional active spaces in the building is simple. It requires instantiating an input sensor and attaching it to the model running in the domain active space.

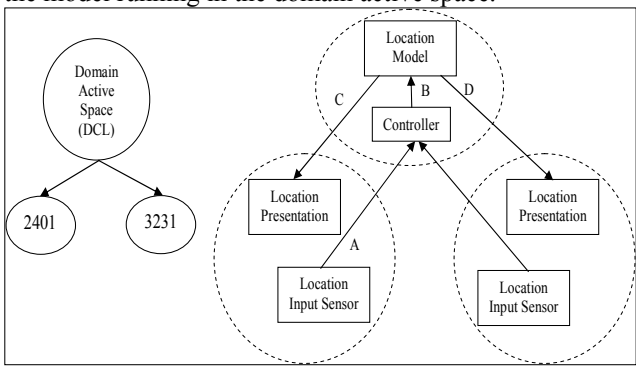


**Figure 8. DCL Active Space hierarchy (left) and corresponding location application instance (right).**

## 3.3 Using the Ticker Tape to Display Location Information

In this section we explain how we use the ticker tape to display user location information. Figure 9 illustrates the ticker tape application and the location application connected by a bridge. The script with the interaction rules is depicted in Figure 10. We describe the functionality based on an example consisting of a user (Andrew) entering an active space (2401).

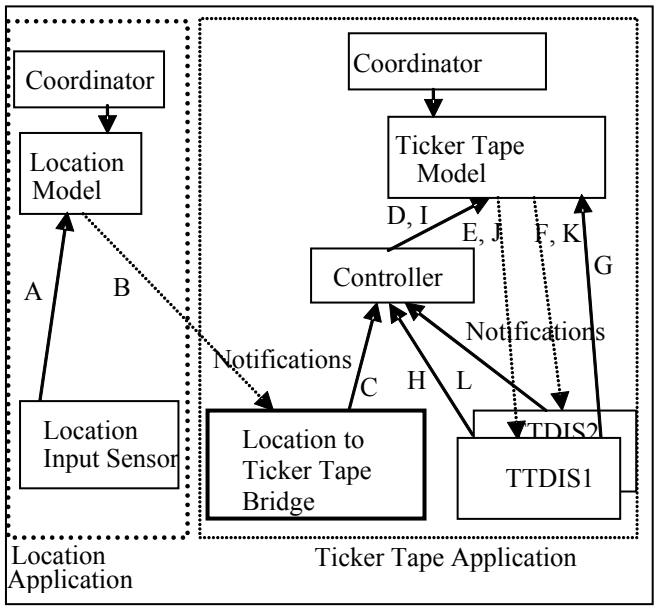


**Figure 9. Using a bridge to display location information in the ticker tape.**

When the user enters the active space, the input sensor of the location application calls a method on the model (via the controller) to report the new user (Andrew) entering active space 2401 (A). The location model updates its data structures to reflect the new location report and notifies all of its listeners with the message "andrew has entered 2401," (B). The location-to-ticker tape bridge parses the username, "Andrew" from the message and calls a method to create a new scroll item in the tickertape with text ("Andrew has entered 2401") and a picture of the detected person (C). The controller receives the message, checks for a mapping, and since no mapping has been defined, it simply forwards the request to the Ticker Tape Model (D). The ticker tape model stores all the fields for the scroll item and notifies all listeners that a new scroll item is available for display on the first display according to its internal display list. The model sends a notification containing a string with the index number of the new item and the id of the ticker tape display input sensor (E, F). The id assigned to the input sensor in the forefront of the figure (TTDIS 1) matches the one included in the notification, so the input sensor calls a method on the ticker tape model to retrieve the scroll item fields (G). Next, the input sensor renders the item using the attributes contained in the item structure and scrolls it across the display. When the scroll item reaches the left side of the display, the input sensor calls a method on the controller to notify that the next input sensor has to begin displaying the item (H). The controller receives the message and forwards it to the ticker tape model (I). The Ticker Tape Model notifies all listeners with a message containing the display id of the next input sensor in the model's internal display list (J, K). This time, TTDIS 2 has the correct id, so it calls a method on the Ticker Tape Model and follows the same steps as the previous input sensor.

Lines 2-6 in the script depicted in Figure 10 parse the location model's notification and extract the name of the person entering or leaving a space. Line 7 sends a request to the ticker tape model (via the controller) to create a new scroll item consisting of text (the source event) and a picture (the name of the file matches the name of the user).

```
1. function(targetController,  sourceModel,
                    sourceEvent)
2.   local pos = strfind(sourceEvent," ")
3.   name = ""
4.   if (pos~=null) then
5.      name = strsub(sourceEvent,1,pos)
6.   end
7.   targetController:defaultSetItem(sourceEvent,
                                          name+".jpg")
8.   end
```

**Figure 10. Location to ticker tape application bridge script.**

## 4. Bridge Creation and Utilization

Bridge creation and utilization are two different tasks and are intended for different people. Creating a bridge requires technical knowledge, including the LuaORB scripting language, information about the source application's model notifications, and the interface of the target application's model. Therefore, this task is reserved for application developers. On the other hand, using a bridge simply requires selecting an entry from a list of available bridges, and therefore does not require technical knowledge.

In order to create a bridge, an application developer writes the application bridge script, and then uses a tool (GUI) to register it. The registration process asks the developer information about the bridge (a description of the functionality the bridge implements), source and target application types, and optional source and target application names. The tool creates a file with a header including the information provided by the user, and the script code. Next, it uses the Gaia Context File System to make it available to the active space.

In order to use the bridges, we provide a tool that lists all available bridges. When the user selects one, the application retrieves the description (stored by the developer's GUI) and presents it to the user. Furthermore, based on the source and target application types (stored also by the GUI), the application presents a list of

compatible applications running in the space. If the developer provided application names, the tool automatically highlights the applications (if they are present). The end user has to select two applications from the list, or accept the pre-selected applications. The tool also provides information about currently running bridges, and allows users to disable them.

We currently have fifteen bridges. Our current bridging mechanism monitors both source and target applications. If any of them crashes, the bridge disables itself automatically.

As part of our future work we plan to extend the end-users tool with functionality to enable bridges automatically based on certain user-defined events such as users entering or leaving the space, or new applications being started.

In order to allow end users to define their own bridges, we need to provide additional support (e.g., a "wizard" like tool) that leverages the infrastructure presented in this paper, and automatically generates the bridge script. For example, the tool would display a list of applications currently running in the active space. Users would choose a pair of applications (source and target) and would select the actions to take on the target (methods to call), based on the notifications fired by the source application model. Such a tool requires further research and therefore we leave it for a future paper.

## 5. Related Work

There are a number of projects [5] [6] [7] [8] [13] that provide a software infrastructure for ubiquitous computing environments. The closest to Gaia are BEACH [7] and iROS [8], in that they consider physically bounded spaces such as offices and meeting rooms, and both of them provide low-level functionality. iROS does not provide explicit support for application development and management, instead, they rely on service synchronization using their event heap. BEACH[7] implements application-level functionality and provides an application framework (based on MVC) to support the development of document-based collaborative applications. Our approach provides a generic active space application framework with support for both collaborative and non-collaborative applications. Furthermore, it provides support for inter-application interaction (active space behavior-level functionality), which is not present in iROS and BEACH.

The concept of application bridging is similar to scripting languages such as LuaOrb, which implements language bindings between Lua[14] and CORBA[15], COM[16], and Java. LuaOrb simplifies the coordination of existing components, and therefore supports the development of applications that reuse COTS components. The application bridge reuses existing applications and defines coordination rules among these applications.

Cooperstock et al. [17] propose a software infrastructure to manage computer-augmented environments, including videoconference environments. They mention the difficulty of using these spaces due to the large amount of devices, and propose a system that adapts automatically and reacts to certain user actions. The application bridging mechanism described in this paper provides the tools to customize the reaction of the active space. The framework described by Cooperstock et al. is customized for a specific type of environment, while application bridges can be used in different environments.

## 6. Conclusion and Future Work

In this paper, we discuss the relevance of active space application interaction as a mechanism to customize the behavior of active spaces. We present a mechanism called an *application bridge* to define interaction rules among applications, and describe our experience with a number of applications that use the mechanism. Application bridges do not require modifications of the applications they bridge, and are independent of the functionality implemented by the applications.

Current results show that application interaction provides an effective mechanism to customize the behavior of active spaces. The ability to reuse existing applications unmodified and defining the interaction rules as scripts allows us to easily obtain new functionality by defining different interaction rules. Furthermore, defining new interaction rules is fast and does not require extensive programming knowledge. For example, the bridge to connect the slide show manager to the x10 application was built and deployed in around five minutes, and the script contains around twenty lines of code.

As users of the prototype active space, we have clearly observed a great change since the installation and utilization of the bridges. We perceive the active space as a reactive environment with some well defined behavior. The results are encouraging because there is still room for further experimentation (e.g., AI techniques) and improvement.

As part of the future work, we plan to continue experimenting with new bridges, integrating new applications, providing mechanisms to automate the installation and monitoring of bridges, and building applications to allow non-expert end-users create their own bridges. We plan to develop more sophisticated bridges that leverage the low-level functionality provided by the Gaia OS (e.g., context, presence, and security). All current bridges alter the application domain functionality of the target application (e.g., display items in the ticker tape, and control the lights). We plan to extend our experiments with bridges that interact with the

coordinator of the target application to modify the composition of the application (application meta-level). For example, a bridge between the location and the music application can move the audio from the room speakers to the user's laptop when it detects that the user is not alone, and can move the audio back to the room when everybody else leaves. Finally, and also as part of our future work, we plan to work on how to resolve conflicts among bridges. We currently solve conflicts via social interaction, i.e., having the parties to agree on what bridge to keep (we exploit the collocated nature of user interaction). However, this option is not feasible in environments were people do not know each other. One of the options is to use user priorities to enforce what bridge to keep.

## 7. Acknowledgements

This research is supported by the National Science Foundation grant NSF 98-70736, NSF 9970139, and NSF infrastructure grant NSF EIA 99-72884.

## References

[1] Jason I. Hong and James A. Landay, "An Infrastructure Approach to Context-Aware Computing," *Human Computer Interaction*, vol. 16(4), 2001.

[2] Anind K. Dey, Daniel Salber, and Gregory D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction (HCI)*, vol. 16(2-4), pp. 97-166, 2001.

[3] Mari Korkea-aho, "Context-Aware Applications Survey," Helsinki University of Technology, Helsinki, Internetworking Seminar April 25 2000.

[4] Bill N. Schilit, Norman Adams, and Roy Want, "Context-Aware Computing Applications," Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, 1994.

[5] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer, "EasyLiving: Technologies for Intelligent Environments," Proceedings of Handheld and Ubiquitous Computing (HUC), pp.12, Bristol, England, 2000.

[6] Joao Pedro Sousa and David Garlan, "Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments," Proceedings of IEEE/IFIP Conference on Software Architecture, pp.29-43, Montreal, 2002.

[7] Peter Tandler, "Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices," Proceedings of Ubicomp 2001: Ubiquitous Computing, pp.96-115, Atlanta, Georgia, 2001.

[8] Brad Johanson, Armando Fox, and Terry Winograd, "Experiences with Ubiquitous Computing Rooms," *IEEE Pervasive Computing Magazine*, vol. 1(2), pp. 67-74, 2002.

[9] Manuel Roman, Christopher K. Hess, Renato Cerqueira, Anand Ranganat, Roy H. Campbell, and Klara Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," *IEEE Pervasive*, vol. 1(4), pp. 74-82, 2002.

[10] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments," Proceedings of 9th IEEE International Symposium on High Performance Distributed Computing,Pittsburgh, 2000.

[11] Manuel Roman and Roy H. Campbell, "A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments," University of Illinois at Urbana-Champaign, Urbana, CS Technical Report UIUCDCS-R-2002-2284 UILU-ENG-2002-1728, July 2002 2002.

[12] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy, "Dynamic component gluing across different componentware systems," Proceedings of International Symposium on Distributed Objects and Applications (DOA'99), pp.362-371, Edinburgh, 1999.

[13] Robert Grimm, Janet Davis, Eric Lemar, Adam McBeath, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall, "Programming for Pervasive Computing Environments," University of Washington, Technical Report: UW-CSE-01-06-01, Washington 2001.

[14] Roberto Ierusalimschy, Luiz Figuereido, and Waldemar Celes, "Lua: An Extensible extension language," Proceedings of Software: Practice and Experience, pp.635-652, 1996.

[15] Michi Henning and Steve Vinosky, *Advanced CORBA Programming with C++*: Addison-Wesley, 1999.

[16] Dale Rogerson, *Inside COM*: Microsoft Press, 1997.

[17] Jeremy R. Cooperstock, Sidney, S. Fels, William Buxton, and Kenneth C. Smith, "Reactive Environments: Throwing Away Your Keyboard and Mouse," *Communications of the ACM*, vol. 40(9), pp. 65-73, 1997.