Plan-Based Proactive Computing

by

Gary Wai Keung Look

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 16, 2003

Certified by . . . .
Howard E. Shrobe
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Plan-Based Proactive Computing

by

Gary Wai Keung Look

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

As the field of ubiquitous computing (ubicomp) has matured, the issue of how ubicomp applications *should* make use of all the devices available to them has not received much attention. We address this issue by presenting a plan-based execution model for creating proactive ubiquitous computing applications. Three applications, each from different domains, were built using this paradigm. These applications demonstrate how knowledge of a person's plan can be used to proactively assist that person. This thesis also discusses the benefits this paradigm provides application developers.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist

Acknowledgments

I would like to first thank my advisor, Dr. Howie Shrobe, for the guidance and perspective he so generously gave. I also thank Dr. Kimberle Koile for our brainstorming sessions and all the encouragement she gave as I was writing this thesis.

I also thank my officemates, Aaron Adler, Harold Fox, and Max Van Kleek. I could not have finished this thesis without your input, coding insight, and all the other help with the little odds-and-ends that go into putting together a thesis.

I also thank João Sousa and Miryung Kim. Our collective efforts put Planlet into orbit.

I am indebted to Stephen Peters, Kevin Quigley, and Krzysztof Gajos for their help with Metaglue and their general input on this thesis. I thank Yao Li for contributing code to the Command Post application and John Stedl for the footage from the Mars rovers.

I also thank Al Carlin for editing an early draft of this thesis.

Last and greatest, I thank God for the grace He extended to get me through the day-to-day thesis grind.

Contents

1 Introduction **13**

1.1 The Promise of Ubiquitous Computing 13

1.2 Ubiquitous Computing and Systems Research 15

1.2.1 Resource Discovery 15

1.2.2 Programming Environments for Ubiquitous Computing 16

1.3 Middleware for Ubiquitous Computing 17

1.4 User-Centered Ubicomp Design Principles 18

1.5 The Case for Plan-Based Proactive Computing 19

2 Plans and Planlet **23**

2.1 Plans As A Representation of User Tasks 23

2.2 Visiting Planlet 25

2.2.1 Planlet Constructs 25

2.2.2 Plan Progress in Planlet 26

2.3 Creating Planlet Plans 28

2.3.1 XML Description of Atomic Steps 29

2.3.2 XML Description of Hierarchical Plans 30

2.4 Carrying Out Plans 32

3 Applying Annotations **35**

3.1 Annotations and Proactivity 35

3.2 Annotations and Plan Monitoring 37

4 Plan-Based Applications **39**

4.1 The *AIRE* Demo Manager 39

4.1.1 The *AIRE* Demo Manager in Action 40

4.1.2 Building the *AIRE* Demo Manager 42

4.2 The *AIRE*-space Configurator 43

4.2.1 Scenarios 44

4.2.2 System Architecture 45

4.3 Command Post for Autonomous Vehicles 49

4.3.1 Information Presented by the Command Post 49

4.3.2 Command Post Architecture 52

5 Evaluation **55**

5.1 Plans as an Abstraction for Programmers 55

5.2 Plans as an Abstraction for End Users 57

5.3 Automatic Configuration of Environments 58

5.4 Better Informed Systems 58

5.5 Self-adaptive Systems 59

5.6 When Plans Are Appropriate 59

6 Future Work **61**

6.1 The Evolving Planlet 61

6.2 Reasoning About Progress Through Plans 62

6.3 Perceiving Progress Through Plans 63

6.4 Improving the Existing Applications 64

7 Related Work **67**

7.1 Task-Driven Computing 67

7.2 Model-Based Programming 69

7.3 Cognitive Orthotics 70

7.4 SharedPlans for Collaboration 71

7.5 Alternate Approaches to Proactive Computing 72

7.6 Subsumption Architectures 73

8 Contributions **75**

A Sample XML Description of a Planlet Plan **77**

List of Figures

2-1 A plan to prepare a person's office for a practice talk. 24

2-2 A snippet of XML that describes a hierarchical plan to configure the devices in a person's office for a practice talk. 31

4-1 The *AIRE* Demo Manager. 41

4-2 An example of a plan used by the Demo Manager is shown above. The demos in the Intelligent Room can be given in any order, and any of them can be skipped, at the discretion of the person giving the demo. 42

4-3 The components of the *AIRE* Demo Manager. 43

4-4 In our system, a plan selector chooses the best plan to achieve the user's goal from a plan library. The resource manager then assigns resources to this plan, using resources from its resource database. The resourced plan is then passed to the plan executor to be carried out. The plan monitor informs the plan executor as progress is made through the plan. 46

4-5 The plan selection mechanism. Figure courtesy of Stephen Peters. . . 47

4-6 Vehicle View. 50

4-7 God's Eye View. 50

4-8 Overall Plan View. Steps are color coded to differentiate between those that have already been completed, those that are currently being carried out, and those that are waiting to have pre-requisites met. 51

4-9 Current Step View. 52

4-10 Gantt Chart. 53

4-11 A model-based executive controls a fleet of autonomous vehicles. Three of the views presented by the Command Post are based on the Planlet representation of the plan this executive is carrying out. The other two views show video footage and location information provided by each of the vehicles. 54

6-1 A person interacting with SAM. 64

Chapter 1

Introduction

Computers have become faster, cheaper, and smaller. This increase in speed, coupled with the decrease in cost and size has allowed computation to be inconspicuously placed everywhere. Indeed, Mark Weiser's vision of ubiquitous computing[1] [66] has become reality.

As the field of ubiquitous computing (ubicomp) has matured, one question has arisen: "How do we coordinate and put to use all this computing power at our disposal?" This thesis presents a plan-based execution model, based on standard principles from AI planning and plan monitoring, that would allow programmers a level of abstraction when addressing this question. This model also allows programmers to develop ubicomp applications that would proactively assist users in their tasks.

In this chapter, we present the benefits of ubicomp applications, a broad overview of the research focus that much of the ubicomp community has adopted and then describe how this current focus led to the motivation behind this thesis.

1.1 The Promise of Ubiquitous Computing

Ubiquitous computing is the area of computer science devoted to helping people accomplish tasks by moving computation away from the desktop and into the physical

[1]Ubiquitous computing is also referred to as pervasive computing. In this thesis, we use the former term.

environment. However, with the diversity of today's computing devices (e.g., cameras, microphones, DSPs, plasma displays, etc.), integrating computation with the physical environment has a deeper implication than just hiding a box behind some floorboard.

This shift in where the computation resides and how it is manifest has a number of effects on where people can benefit from computers and what role computers play in people's work. By moving the computer into the environment, the boundaries of the traditional workstation are increased. Computers still support the work done while sitting at the desk in your office, but they can now also play a role in work done at various other places. For example, ubicomp applications help biologists record and retrieve information about their experiments [8], assist the residents of elderly care facilities with their day-to-day activities [55], and allow for the information generated during spontaneous meetings held around the office water cooler to be captured [62].

The increased variety of places where computation can be put to use also brings about new ways of interacting with computer systems. For example, in the non-traditional computing scenarios mentioned above, it is not practical to tether a person to a keyboard. Although the computer is now everywhere, there is no where to place a keyboard. It now becomes the computer's responsibility to use its available sensors to translate the user's intent and other data — as communicated via whatever modality may be convenient to the user, such as speech or gesture — into the zeroes and ones it understands.

By pushing computation into the environment, ubicomp applications free the user from having to struggle with a tool (the computer) that is supposed to make their tasks easier. The user can then focus on the actual task at hand and use whatever devices are available in the environment to facilitate this task. The computing-rich environment can provide additional aid by performing routine chores in support of that task.

1.2 Ubiquitous Computing and Systems Research

To motivate a plan-based execution model for ubiquitous computing, we first situate ourselves in the field by presenting three major research agendas that are being explored. Then, in section 1.5 we point out the "blind spots" not addressed by these research agendas, and how a plan-based execution model meets these shortcomings. A discussion of work that is related to what is presented in this thesis is found in Chapter 7.

The three major research agendas that we discuss are:

1. system architectures, where the driving research question is "What infrastructure do we need in order to allow a ubicomp application to talk to and use all the computing power available to it?"

2. middleware, where the driving research question is "How can we make it easier for programmers to write ubicomp applications?"

3. human-computer interaction (HCI), where the driving research question is "What are the user-centered design principles to keep in mind when building ubicomp applications?"

This section focuses on the first of these areas, and it describes resource/service[2] discovery mechanisms used to coordinate the large number of sensors, actuators, and computers used by ubicomp applications. We also discuss various programming environments that serve as the platforms (i.e., the "ubicomp operating systems") on which ubicomp applications are built.

1.2.1 Resource Discovery

In Sun's Jini discovery mechanism [7], resources provide a handle to themselves through the use of proxy objects. Proxy objects are downloaded by the client to access a service. To allow clients access to their proxy object, a resource registers

[2]A service is some capability, such as text output. Resources, such as an LED display, provide services

its proxy object with Jini “lookup services.” Clients obtain a given proxy object by contacting lookup services and querying for the particular Java proxy object.

Berkeley’s Service Discovery Service (SDS) uses a hierarchical server architecture in which resources register with an SDS server and clients query these servers for services [19]. The service description and query expressions used by services and clients are XML-based.

The Intentional Naming System (INS) [2] uses an XML-like hierarchical naming scheme to describe a service’s properties and functionality. Service requests can be answered using an early-binding option, in which INS returns a list of resources that fulfill the request, but unlike Jini and SDS, INS also offers a late-binding option, in which INS forwards the service request and any necessary data to the best available service.

1.2.2 Programming Environments for Ubiquitous Computing

Gaia [59] provides ubicomp developers with an infrastructure that treats spaces with embedded computational power as programmable spaces. Gaia applications are built using a generalization of the Model-View-Controller (MVC) paradigm. The components that make up the MVC description of a Gaia application are generically described and then are specifically instantiated for a given space by consulting a data repository that contains a description of the available resources in that space.

Aura [29] adopts a more user-centered approach to ubicomp applications than Gaia. Aura’s main research thrust is to minimize the user distraction associated with device configuration and varying levels of service availability encountered by mobile users as they move from one area to another. To this end, Aura introduces the notion of a user task, which is represented by a collection of services. A user’s computational *aura* then moves with him and uses a resource discovery mechanism based on Jini to perform a sort of “cyber-foraging” to ensure that the necessary services are available for the user to continue his task.

one.world [31] uses a resource management scheme that returns *leases* to limit the duration which an application has access to a requested service. While leases could be renewed, the fact that there is a possibility a lease won't be renewed forces the programmer to choose some sort of adaptation strategy to account for change in the computing environment. To further highlight the notion of dynamism in ubicomp applications, one.world allows applications to change their behavior at runtime by dynamically adding or removing event handlers, migrate from one host to another, and checkpoint application state.

Finally, Metaglue [17] is a distributed agent system in which applications and the resources they use are represented as agents. Metaglue provides on-demand agent startup and a persistent storage mechanism. Together, these features essentially mean that Metaglue agents are always available for use. The Metaglue resource manager [28] assigns resources to requesting agents based on a utility function that takes into account the estimated benefit of using that resource, as determined by the preferences a user has between the different qualities of service provided by that resource, and the cost of taking that resource away from the agent that is currently using it.

1.3 Middleware for Ubiquitous Computing

Whereas Section 1.2 discussed work that lays down the infrastructure to make ubicomp applications possible, this section describes tools that make it easier to build ubicomp applications.

The most notable example of middleware for ubiquitous computing is the Context Toolkit [22]. Anind Dey defines context as "any information that can be used to characterize the situation of entities (i.e., whether a person, place, or object) that are considered relevant to the interaction between a user and an application." Context can be used to help determine what the appropriate behavior for a ubicomp application should be in different situations, and the Context Toolkit provides a formalized set of abstractions to handle the acquisition of and response to, different pieces of context. The Context Toolkit has been used to build a number of context-aware applications including a conference assistant that directs attendees to sessions that would be of interest to them [23] and an intercom system that delivers messages to people at their current location only if they are in a situation where they can be interrupted by the message [22].

As a recent survey of location systems for ubiquitous computing has shown, location is an important piece of contextual data that has been the focus of much research [32]. Some of the most well known location systems are the Global Positioning System (GPS), Active Badge, and the Cricket Location System. A few of the differences between these systems are: GPS is used outdoors whereas Active Badges and Crickets are used indoors; and GPS and Cricket receivers listen for signals which are then used to triangulate a user's location, whereas Active Badges announce a user's identity to a sensor network in order to locate the user (at the loss of privacy on the Badge-wearer's part).

The differences between these three location systems illustrates the need for a principled approach when dealing with location information. One such approach is the location stack [33], a layered software engineering model that calls for the abstraction of higher-level concepts of location from low-level sensor data. This model, similar to the seven-layer Open System Interconnect model for computer networks, could be thought of as a structured way of using the location widgets from the Context Toolkit. A second approach to dealing with the heterogeneity of location information is illustrated in PLACE [40], which uses a semantic representation of location to fuse location data from different sources.

1.4 User-Centered Ubicomp Design Principles

Research from the field of human-computer interaction (HCI) is the third major area of ubicomp research. Work in this area seeks to understand how people like to work in order to come up with principles for designing useful and useable applications.

The Pebbles project [45] is an example of how PDAs and other hand-held computers can be used to provide another interface to PCs, thereby augmenting the ways in which desktop computers are used. Eventually, Pebbles would like to enable PDAs to serve as a sort of universal, portable controller for all sorts of devices.

While the goal of Pebbles is to enable one device to control many others, the iCrafter system [56] allows individual services offered by Stanford's Interactive Room [37] to be controlled by many other types of devices. To make this possible, the interfaces provided by iCrafter account for the display and modal capabilities of the requesting device (e.g., the interface provided to a PalmOS-enabled device would need to account for the lack of screen real estate and the fact that PalmOS does not support drag-and-drop).

In addition to the HCI work on displaying interfaces for different devices, there has been research on developing multi-modal interfaces to supplement or replace the use of keyboard and mouse. Bolt's [10] "Put That There" paper was one of the first papers in this area. Bolt demonstrated a system where users were able to manipulate objects on a screen by pointing at them and then referring to them later using pronouns. More recent projects include QuickSet [50] and Assist [3], which seek to integrate speech and sketching as alternate input modalities; Alice [18, 51], which looks at how virtual reality could be used as an alternate modality; and Carnegie Mellon's Command Post of the Future [44], which seeks to integrate speech, handwriting, gestures, and gaze tracking into a group collaboration tool.

1.5 The Case for Plan-Based Proactive Computing

From the previous sections, we see how recent work in ubicomp research has produced resource discovery mechanisms, programming environments, middleware, and user interfaces that simplify the creation of user-centered applications that adapt to the different resources available in various computing environments. However, despite these advances, none of the research so far discussed has addressed how a ubicomp application *should* use all the sensors, actuators, and other computing devices available to actually help a user carry out a task. Due to this shortcoming, many ubicomp applications operate in a myopic mode that lacks (1) the ability to proactively assist the user in his task, (2) the ability to inform the user when he is unfamiliar with the task, and (3) the ability to gracefully failover (and explain the rationale for this behavior) when resources become unavailable.

David Tennenhouse made a similar observation when reflecting on the advances in computing power that have made ubiquitous computing possible [61]. Due to the vast number of devices that are available for people to use and the speed with which they respond to things, Tennenhouse proposes the need for new abstractions to better manage the behavior of these devices as they attempt to help us in our everyday lives.

Given this need for ubicomp applications to be smarter in the way they assist their users, this thesis proposes a layer of middleware, called Planlet, to support a plan-based execution model for ubicomp applications, and discusses the lessons learned from building ubicomp applications using such a model. While early work in this area has proposed a task-driven model [16, 65] to ubiquitous computing, these approaches have treated a task as a set of applications, and progress through the task as application state. However, applications are only the tools used to carry out a task; they don't necessarily represent the task itself.

Rather, behind every task, the user has some goal in mind that would be obtained by completing the task, and each step in the task (which may require a tool to complete) contributes to attaining that goal. In this thesis, we claim that by presenting ubicomp applications with an explicit plan-based representation of a user's task — a representation that includes the steps involved in the task, the resources needed by each task, and the causal and data flow relations between those steps — it will be easier to develop ubicomp applications that demonstrate a degree of *proactivity* that current ubicomp applications do not possess.

In particular, the idea of plan-based proactive computing provides the following benefits:

- A higher level of abstraction for designing ubicomp applications. From the perspective of application developers, working with a plan-representation of a task allows for a decoupling of application behavior from the actual implementation of that behavior.

- A higher level of abstraction for end users to communicate intent. End users can present applications with the goal they want to achieve, instead of the individual actions that they will perform. The application could then produce the plan that would achieve that goal and use this plan to guide its interaction with the user.

- Automatic configuration of environments. By knowing what the user plans to do, an application could reserve necessary resources and configure devices the user may soon need.
- Better informed systems. Given an explicit representation of a particular task, an application could guide users through the steps of that task if they are unfamiliar with it. If there are multiple things in the task that a user could tend to, such a system could also help direct the user's focus to the task that requires his attention most.
- Self-adaptive systems. If an application detects a component failure, it can use its representation of the user's plan to suggest an alternate course of action that would still achieve his intended goal.

While the focus of this thesis is on the benefits that ubicomp applications and their designers achieve by adopting a plan-based programming paradigm, we acknowledge that there are a number of closely related topics that merit mention. In particular, the topics of planning [67] and plan recognition [5], are relevant and complement this work, but incorporating planning algorithms to generate user plans or using plan recognition techniques to infer user intent is beyond the scope of this thesis.

To manage the scope of this thesis, we will assume a pre-compiled plan library as the source of plans for ubicomp applications, and allow for plans to be created and added to this library. We will also assume that the user explicitly indicates his goal to the application, as opposed to having the application infer the user's goal from observing his actions. These simplifying assumptions allow us to focus on studying the benefits of plan-based proactive computing.

Chapter 2

Plans and Planlet

This chapter formally defines what information a plan contains and how ubicomp applications can be designed using a plan-based programming paradigm. We also introduce Planlet, a software middleware layer that can be used by plan-directed applications to represent plans and user progress in these plans.

2.1 Plans As A Representation of User Tasks

A plan is a decomposition of a task into a set of partially-ordered steps. Each step has a set of pre-conditions that must be met before the step can be performed, and a set of post-conditions that hold after the step has been performed.

The partial ordering of the steps in a plan are defined by a set of data flow links, causal links and ordering constraints. A data flow link between step X and Y represents any material, whether physical or information, produced by X and used by Y. A causal link, represented as $X \xrightarrow{c} Y$, joins the post-condition of a step X with pre-condition c of another step Y, and can be interpreted as "step X achieves precondition c required by step Y". An ordering constraint between steps X and Y, denoted by $X \prec Y$, represents the fact that step X precedes (although not necessarily immediately) step Y; ordering constraints thus represent control flow in a plan and are used to prevent race conditions where performing one step would clobber either the effect of performing the other step or the pre-condition enabling the other step.

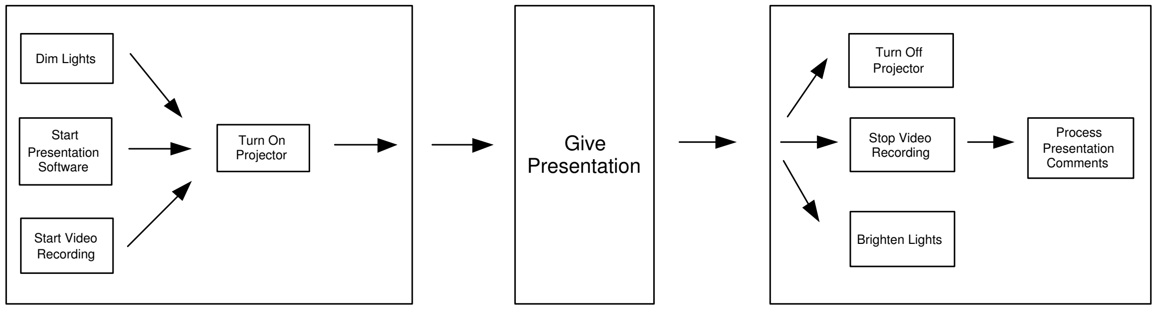


Figure 2-1: A plan to prepare a person's office for a practice talk.

Each step in a plan may require a set of resources subject to certain constraints. For example, a plan step might require a sufficiently bright light source for reading (e.g., the halogen lamp on my nightstand). Often there is more than one resource capable of meeting the plan's needs (for example, the incandescent ceiling lights also provide enough light for reading). In general, the choice of which resources to use is late bound to account for the actual operating conditions at execution time.

In addition, the plan may include hierarchical substructure. Substeps of the plan may either contain explicit internal plans, or they may be only the statement of a goal, requiring the selection of a plan capable of realizing that goal. As with resource allocation, this decision may be bound late, allowing current conditions to influence the choice.

Both the plan as a whole and the individual steps in the plan have a set of annotations. An annotation can be any information that is relevant to carrying out a plan, such as: which plan steps require special attention because they are more likely to fail; recommended orderings of steps, modulo the partial-ordering imposed by the plans causal and ordering links; instructions on how to use a service; and estimated service qualities provided by carrying out the plan (e.g., a plan may accomplish a task with a high degree of privacy, but in a very slow manner). How applications use annotations to proactively assist their users is described in Chapter 3.

Plans can be represented graphically with a directed acyclic graph. A plan to prepare a person's office for a practice talk is presented in Figure 2-1.

2.2 Visiting Planlet

To provide ubicomp applications with a way to represent user plans and manage progress in these plans, we present Planlet, a generic middleware layer implemented in Java. Applications query Planlet for what steps the user or the application could perform next and then the application uses this information to take appropriate action. As steps are completed, the application informs Planlet, and Planlet updates its model of the current plan state. A full discussion of Planlet can be found elsewhere [38]; in the next few sections, we briefly describe how Planlet implements the plan representation described in Section 2.1 and manages progress through these plans.

2.2.1 Planlet Constructs

Planlet represents a plan as a directed acyclic graph, where the nodes are either subplans or (domain-defined) atomic steps, and the edges represent the flow of materials and any ordering constraints between nodes. Both plans and steps define a set of inputs and a set of outputs; directed flows between outputs and inputs define the plan's graph (see Figure 2-1 for an example).

Plans and steps are annotated with property expressions and context information, which can be updated as a plan is being executed. Generic descriptions of the resources needed by a plan (e.g., a "display" resource) are also recorded as annotations. In the current version of Planlet, expressions can evaluate to either strings or numbers, and context information is stored as a string. Future versions of Planlet will be able to support more advanced data types, but in our initial experiments, we found that these data types were sufficient. Applications take advantage of these annotations by using them to provide proactive assistance or clarify user confusion.

A hierarchical plan contains a collection of related subplans, and it allows for plans to be reused in other plans by abstracting the aggregate work done by the hierarchical plan's subplans. Inputs to the subplans contained in the hierarchical plan are defined either as an input to the hierarchical plan or as an output of a subplan contained in the hierarchical plan. The subplans in a hierarchical plan do not have to be bound at the time the plan is defined. Planlet allows for plans to be changed as they are carried out; thus, a hierarchical plan can contain steps that only specify a goal. At execution time, this goal is then resolved to a specific plan, as determined by the conditions at that time.

In addition to the atomic step and hierarchical plan, Planlet also provides two convenience constructs that can be used to define a plan. The first of these constructs is the LoopPlan, which is used to denote repetition of a step bound to a certain condition, similar to the 'while' construct in a programming language. A LoopPlan is defined with a subplan that is repeated and a conditional expression that determines when the iteration terminates. Internally, Planlet manages iteration via recursive definition (i.e., "perform the first iteration, and then (recursively) repeat; stop if the loop condition is false).

The second convenience construct is the MultiPlan, which is used to denote multiple instances of the same plan. This is similar to the collection construct in programming languages. MultiPlans are used to represent batch operations such as running the same test on a collection of distinct samples in a biology lab.

Lastly, in addition to supporting dynamic modification of a plan, Planlet also allows for control flow to be decided at the time a plan is carried out. This is achieved by annotating the edges between plans with a conditional expression that is evaluated either true or false during the plan's execution. The producer plan informs the downstream consumer that the value it needs is available if and only if the condition is evaluated true.

2.2.2 Plan Progress in Planlet

The assumption underlying Planlet is that the role of ubicomp applications is to assist rather than control the user in carrying out a plan. A plan and the progress through it represent knowledge about events in the real world. These events are beyond the control of, and, in many cases, beyond accurate monitoring by computer systems. When automatic context sensing is limited, how much Planlet knows about the progress through the plan is very much dependent upon information from the user.

Therefore, Planlet does not assume or rely on a 100% up-to-date replica of the state of materials in the real world. Planlet does keep notes of which state the materials should be in, according to the plan, and informs the application/user of that. Nevertheless, ubicomp applications can take a range of measures upon noticing that the user is deviating from the plan. Those measures may range from making sure the user is aware he is deviating (and allow or even register the new user actions as an alternative branch in a known plan), to refusing acknowledgment of completion of anything other than the permitted steps (akin to workflow systems), to actuation in the physical environment or communication with authorities whenever safety may be compromised (e.g., when monitoring Alzheimer's patients).

To model the progress through a plan, each node in a Planlet plan graph can be in one of four states. The first state, Waiting, describes a plan has not been executed yet and is not ready to be executed. The second state, Candidate, means that a plan has not been executed yet but is ready to be executed. The third state, Done, means that the user explicitly indicated that the plan was completed. The fourth state, Eliminated, means that this plan instance has been eliminated from the set of instances with Candidate status without the user's explicit indication. The Eliminated and Done states differ in that the Done state is determined by the user's explicit completion indication whereas a plan instance is inferred to be Eliminated because some other plan dependent on the Eliminated one is marked Done.

Formally, an atomic step is in the Waiting state if and only if there exists some input that is not ready. A step instance is in the Candidate state if all of its inputs are ready and the step has not been performed yet. Thus, the step instances in the Candidate state form the set of next possible actions that a user can perform. Step instances are in the Done state if and only if their outputs are produced as indicated by an explicit user completion notification. Step instances enter the Eliminated state if and only if all their outputs are produced and this fact is inferred from the plan graph, rather than being explicitly indicated by the user.

For the other types of plan instances, they are identified as being in the Waiting state if and only if there exists some subplan in the Waiting state and there is no subplan in the Candidate state. They are in the Candidate state if and only if there exists some subplan in the Candidate state. They are in the Done state if and only if all subplans are in the Done state. Lastly, they are identified as Eliminated if and only if all subplans are either in the Done or Eliminated state.

Since it may not be possible to record the completion of every step, Planlet has two modes of marking a step as done: soft_complete and hard_complete. When soft_complete is called on a step or subplan, Planlet checks if that step is on the list of steps that the user is expected to take next. If not, it returns a diagnostic and does not change the state of the plan. This diagnostic is examined by the application and an appropriate measure can be taken. If the application is able to determine that enforcing the completion notice is allowable from the user's point of view, it can call hard_complete on that same step or subplan. For a hard_complete, Planlet assumes that the user did not bother to fill in all the intermediate details of what he is doing — and that the application allows that — so it just marks the step as being Done (albeit done unexpectedly because it is a step that is dependent on another step that has not been marked as such). Planlet then advances the plan execution state to the indicated point.

2.3 Creating Planlet Plans

Planlet relies on an outside source, such as the ubicomp application itself, to obtain the plan. Depending on the domain, that outside source may construct a plan given a goal (AI planning); it may learn plans by observation of behavior patterns (machine learning); or it may rely on a user who is a domain expert to define the plan (plan editing). Planlet is not concerned with how the plan definition is obtained — it exposes an API [38] that enables external components to create and modify a plan.

In addition to providing an API for programmatic access to plan editing, Planlet is also able to take an XML description of a plan and convert that into a Planlet plan. We discuss the structure of the XML document in the remainder of this section. A discussion of the lessons learned from providing this feature is given in Section 5.1.

2.3.1 XML Description of Atomic Steps

An XML snippet that describes the atomic step of starting a video recording (one of the steps in the example plan shown in Figure 2-1) is shown below:

```
<step>
  <name> Start recording video </name>
  <numinputs> 0 </numinputs>
  <numoutputs> 1 </numoutputs>
  <properties>
    <property name="resource" type="String" value="video camera"/>
    <property name="method" type="String" value="record"/>
  </properties>
</step>
```

The `<step>` tag marks the beginning of the step's XML description. The tags in a step's description are fairly self-explanatory:

- `<name>`: the name of the atomic step
- `<numinputs>`: the number of pre-conditions and ordering constraints that need to be fulfilled before this step is ready to execute
- `<numoutputs>`: the number of post-conditions and ordering constraints that are satisfied after this step is completed
- `<properties>`: marks the beginning of the list of annotations belonging to this step. Each annotation is denoted by a `<property>` tag
- `<property>`: this tag is used to denote an individual annotation. The `<property>` tag's attributes indicate the name of the annotation, the annotation's data, and the annotation's data type

2.3.2 XML Description of Hierarchical Plans

A partial example of an XML file describing the hierarchical plan to configure the devices in a person's office for a practice talk (i.e., the left-most box of the three large boxes in Figure 2-1) is shown in Figure 2-2. The complete file can be found in Appendix A.

A hierarchical plan's description is encapsulated by `<hierarchicalplan>` and `</hierarchicalplan>` tags. The same set of tags used to describe atomic steps (see Section 2.3.1) are also used to describe a hierarchical plan. In addition to these tags, the XML description of a hierarchical plan uses two other sets of tags to store information about a hierarchical plan's subplans and the links between these subplans:

The list of subplans is indicated by `<subplans>` and `</subplans>` tags. The content between these two tags are the XML description of the subplans; i.e., atomic steps and other hierarchical plans.

The links between subplans are contained between the `<flows>` and `</flows>` tags. The description of an individual link is encapsulated between `<flow>` and `</flow>` tags. The tags used to specify a link from plan X to plan Y are:

- `<producerplanname>`: the name of the plan the link is coming from (i.e., X). We'll refer to this plan as the producer
- `<producerinouttype>`: describes whether this link is coming from an output of the producer (as is usually the case) or coming from an input of the producer (which happens when a hierarchical plan routes one of its inputs to a subplan)
- `<producerportnum>`: indicates which output (or input, depending on the contents of the `<producerinouttype>` tag) this link is coming from
- `<consumerplanname>`: the name of the plan the link is going to (i.e., Y). We'll refer to this plan as the consumer
- `<consumerinouttype>`: describes whether this link is going to an input of the consumer (as is usually the case) or going to an output of the consumer (which happens when a link is routed to the output of a hierarchical plan)

```
<hierarchicalplan>
  <name> Setup devices for practice talk </name>
  <numinputs> 0 </numinputs>
  <numoutputs> 1 </numoutputs>

  <subplans>

    <step>
      <name> Start Recording Video </name>
      <numinputs> 0 </numinputs>
      <numoutputs> 1 </numoutputs>
      <properties>
        <property name="resource" type="String" value="video camera"/>
        <property name="method" type="String" value="record"/>
      </properties>
    </step>

    <step>
      <name> Turn On Projector </name>
      <numinputs> 3 </numinputs>
      <numoutputs> 1 </numoutputs>
      <properties>
        <property name="resource" type="String" value="projector"/>
        <property name="method" type="String" value="on"/>
      </properties>
    </step>

    ...

  </subplans>

  <flows>
    <flow>
      <producerplanname> Start Recording Video </producerplanname>
      <producerinouttype> output </producerinouttype>
      <producerportnum> 0 </producerportnum>
      <consumerplanname> Turn On Projector </consumerplanname>
      <consumerinouttype> input </consumerinouttype>
      <consumerportnum> 2 </consumerportnum>
    </flow>

    ...

  </flows>

</hierarchicalplan>
```

Figure 2-2: A snippet of XML that describes a hierarchical plan to configure the devices in a person's office for a practice talk.

- `<consumerportnum>`: indicates which input (or output, depending on the contents of the `<consumerinouttype>` tag) this link is going to

2.4 Carrying Out Plans

A plan-based application needs to be able to select a plan that best matches the user's task and then assign resources to this plan. The application then carries out this "resourced plan." In this section we focus on the two pieces of functionality an application needs to provide in order to carry out a resourced plan: a plan executor that actually carries out steps of the plan, and a plan monitor that watches over the progress of the plan. We present one way of selecting plans and assigning resources to them in Section 4.2.2.

The plan executor interprets the plan step and any annotations the step may have and then dispatches the necessary information to the software components that can complete the step. In the Planlet-based applications we built, this means that our plan executor dispatches (Planlet) steps in the Candidate state. Steps that the application carries out have the following annotations: a generic resource description (e.g., a "display" resource), the name of a method offered by the service (e.g., "show") and a list of arguments to pass to that method (e.g., the name of a presentation). Our executor then calls that method of the specific resource that has been assigned to the plan.

While interpreting the plan and determining what steps are ready for dispatch, it is necessary to consider the fact that 1) a person, and not the application, may be the one who is performing a particular task (as noted in an annotation) and 2) a step may not complete successfully, regardless of who is responsible for performing that step. For these reasons, it is necessary to have a plan monitoring component.

If a user is the one who is performing a task, the plan monitor must be able to determine when he is finished with the task. Approaches to making this determination, include using perceptual techniques to observe and infer that the user is done with a step, or adopting turn-taking or discourse modeling approaches to indicate when a user is done with a step. At the moment, we adopt a simpler approach to plan monitoring and listen for explicit user cues to indicate that he has completed a task, such as a button press or a vocal statement like, “I’m done.”

To support monitoring for failure, plans have steps that test if the state of the world is within expected bounds. If this is not the case, control can be passed to a diagnosis and recovery unit to remedy the problem. Since the application already has a plan model of the task carried out by the user, model-based diagnosis is a natural approach to use to diagnose failures. Diagnosis and recovery is beyond the scope of this thesis, but we discuss it further in Section 5.5.

Chapter 3

Applying Annotations

The work in this thesis stems from research in the AI community covering plan monitoring [25, 63] and robotics [26]. The Planlet middleware introduced in Section 2.2 furthers this work by making annotations available for applications to use. A number of different components, each making use of these annotations, can be layered on top of Planlet. These components can augment the core Planlet capability of modeling plan progress by providing the following functionalities: determining what proactive measures to take, carrying out diagnosis and recovery when a plan goes awry, and inferring when plan steps complete. How these components would use annotations is presented below. These ideas came from our experiences with building a number of plan-based applications; these applications are described in Chapter 4.

3.1 Annotations and Proactivity

A proactive reasoning component can be used by plan-based applications to identify actions that would meet a person's needs before he consciously realizes them. Such a component can use annotations to provide three different types of proactive measures: selecting plans appropriate to the service qualities that are important to a person; automatically configuring devices to reduce user distraction; and guiding or pre-empting the need for diagnosis and recovery.

Carrying out a plan can be regarded as an incremental decision making process [9, 63] between different plans that achieve the same goal. Oftentimes, the person's choice of plans is determined by the different service qualities offered by the alternative plans. The level of service qualities such as speed, reliability, visibility, and intrusiveness, can be easily represented as an annotation to a plan. They can then be used to select which plan would provide the highest utility to the person carrying out the plan based on his current service requirements. Section 4.2.2 provides a detailed description of how plan selection based on service qualities is performed.

Annotations can also list the resources used in a plan and the location and preferred means by which these resources are used. With these annotations, the proactive reasoning component can contact a resource manager and reserve devices according to the preferences given in the annotation, similar to how service qualities can be used to select different plans. For example, annotations can be used to indicate that a person wants a resource that is able to present important messages inconspicuously. The resource manager could then provide the plan with a scrolling LED sign visible only to the person expecting the message, or if no such sign is available, provide a software agent that would send a message to the person's mobile phone, with the drawback being that this would result in a less inconspicuous way of checking the message. Annotations can also be used to set devices to a preferred state; for example, if a plan calls for a reading lamp, the proactive reasoning component can use an annotation to adjust the brightness to a preferred level.

Lastly, annotations can be used to identify steps along the critical path of a plan. Determining what these steps are can be done in a number of ways: analyzing a Pert chart view of the plan, using heuristics taken from the plan domain, and using empirical data are all ways of identifying what steps of a plan are likely to fail. Once a plan's "pressure points" are thus identified, these annotations can be used to facilitate diagnosis and recovery by going through all the annotations to generate a set of candidates that could have contributed to the failure. In fact, since information about the fragility of plan steps is available even before a failure occurs, a plan monitoring component can use these annotations to warn a person of common points of failure as he is carrying out the plan. Also, due to the directed, acyclic graph representation of plans, a straight-forward graph traversal identifies what other steps are at risk if a particular step does not go as planned. By providing this information, plan annotations actually pre-empt some of the need for diagnosis and recovery, since people can be made aware of problematic parts of a plan.

3.2 Annotations and Plan Monitoring

For the reasons described in Section 2.4, plan-based applications require a plan monitoring component to update plan status. In many systems, however [27, 30], the set of monitoring actions is hard-coded. The obvious drawback to this approach is that if a plan changes during execution time, accurate monitoring may not be possible. Another monitoring approach is to have a person acknowledge the completion of each step in his plan; while this is an accurate technique, it is often too distracting and burdensome for the individual.

The use of annotations, along with Planlet's ability to late-bind a plan, allows for a much greater degree of flexibility in the plan monitoring scheme used by plan-based applications. Similar to rationale-based monitoring [53, 54], the annotations in a Planlet plan can be used to store information about the pre- and post-conditions of each step in that plan. This information can then used to monitor when a step has either started or completed.

For example, in many domains (e.g., the biology lab or a wood-working shed), the current plan step a person is working on is closely tied to his location. By including this information as an annotation, a plan monitor could infer that a step has started because the user has entered a particular area (e.g., a wood crafter is about to shape the legs of the table he is making because he walked to the corner where his lathe is located).

To handle the fact that relevant monitors can change, we propose using a blackboard architecture [46, 47] as the source of information about world events. This would eliminate the need to specify particular monitors *a priori* because the plan monitor could use the plan's annotations to dynamically decide which blackboard knowledge sources to subscribe to. An added benefit of using a blackboard is that it supports the fact that the pre- and post-conditions specified in the annotations can be at any level of abstraction (e.g., going back to the woodshop analogy, the annotation for one step may state that the step has completed when the wood crafter leaves his lathe while another step may be considered complete when a digital level has a zero reading). Finally, since this approach does not require a person to explicitly acknowledge completing each step (although it does support it), the individual would not have to be distracted with that burden.

Chapter 4

Plan-Based Applications

A number of ubicomp applications were written to show the benefits of a plan-based execution model. Although these applications span a number of different domains, each of them illustrate a number of the benefits mentioned in Section 1.5. These applications were all built using Planlet, and modeled according to the architecture described in Section 2.4. The applications were written (or currently being rewritten) using Metaglue [17], one of the programming environments for ubiquitous computing described in Section 1.2.2. This choice was made due to the conveniently large amount of available Metaglue support; any of the other programming environments described in Section 1.2.2 could have been used as well.

4.1 The *AIRE* Demo Manager

There has been a lot of work done on context-aware tour guides that allow a person to learn about a new place without the need of a human guide [1, 14, 15]. However, there are still times (such as when sponsors visit research labs) when it is necessary to provide visitors with human-guided tours. This is an especially common situation in our research group, named *AIRE*[1]. *AIRE*'s research focus is on building intelligent environments; this focus includes a wide assortment of research topics ranging from the communication substrate needed to coordinate agents (Metaglue), to multi-modal

[1]*AIRE* [4] stands for Agent-based, Intelligent, Responsive Environments.

and perceptual interfaces [3, 21, 48, 62]. The *AIRE* group has built a number of intelligent environments (dubbed *AIRE*-spaces) including meeting rooms, offices, and public spaces.

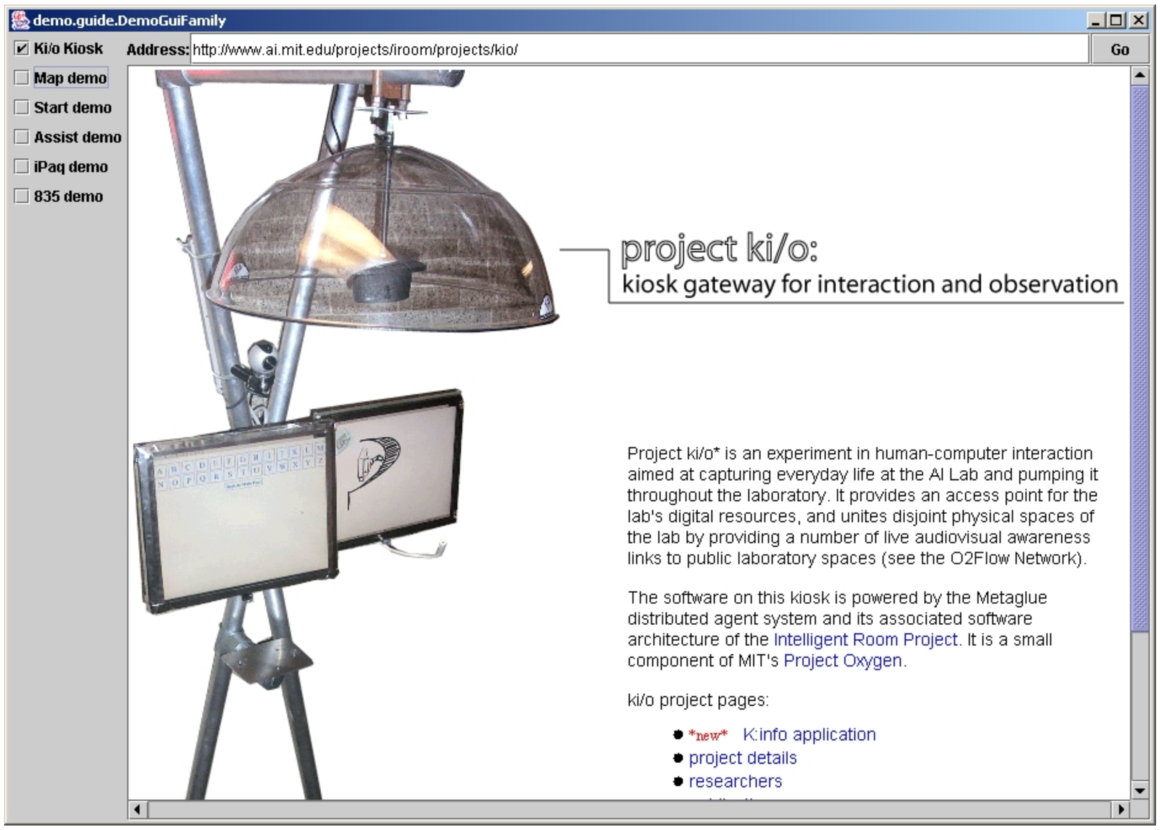
There are hundreds of software components running in our different *AIRE*-spaces. The large degree of heterogeneity between these agents, and in the nature of the spaces themselves, makes it difficult to give a structured, cohesive, and comprehensive demonstration of our research. To help group members give better demos of our research[2], we created the *AIRE* Demo Manager.

4.1.1 The ***AIRE*** Demo Manager in Action

Building on the earlier work of the AFAIK Help System [13], the *AIRE* Demo Manager displays an agenda of the items in the demo to both the person leading the demo and to the visiting group. The person leading the demo can choose a particular demo agenda from a library of possible demos, depending on the audience and the available time. In its main display panel, the Demo Manager displays information (such as the HTML help files generated by AFAIK) about the current item on the demo's agenda. The demo leader can use this information to structure his presentation in one of two ways. He can use it as a rough outline to guide his presentation, elaborating, as necessary, on certain points which are introduced by the Demo Manager, or the demo leader can rely more heavily on this information when presenting the current research project to be demonstrated (e.g., using this information as a quick reference guide to running that particular demo). Usually, both approaches are used.

As the demo progresses, the Demo Manager checks off the items on the agenda that have already been shown and updates the information displayed on its main panel. To provide a degree of naturalness in the flow of the demo, this update is done implicitly according to verbal cues given by the demo leader. Of course, an explicit, "I'm done" verbal command is also available.

[2]In this section, we will use the term demo to refer to the set of projects shown, an individual project in that set, and the act of giving a demonstration. The meaning of the term will be clear from context.



project ki/o:
kiosk gateway for interaction and observation

Project ki/o* is an experiment in human-computer interaction aimed at capturing everyday life at the AI Lab and pumping it throughout the laboratory. It provides an access point for the lab's digital resources, and unites disjoint physical spaces of the lab by providing a number of live audiovisual awareness links to public laboratory spaces (see the O2Flow Network).

The software on this kiosk is powered by the Metaglue distributed agent system and its associated software architecture of the Intelligent Room Project. It is a small component of MIT's Project Oxygen.

ki/o project pages:

- *new* K:info application
- project details
- researchers

Figure 4-1: The *AIRE* Demo Manager.

Since demos of our research take visitors across a number of different *AIRE*-spaces, the Demo Manager display also migrates with the demo group, and reappears on a display at the next location in the demo. The Demo Manager's display is shown in Figure 4-1.

In addition to presenting information, the Demo Manager will also start up software components that are used in the next part of the demo. Another proactive measure that the Demo Manager takes is keeping track of how long each segment of the demo takes. When a particular segment runs long, the person leading the demo is notified of this fact. At the moment, the default way of making this notification is to highlight the late-running demo on the Demo Manager's agenda, but future implementations will take more discreet means, such as vibrating the demo leader's cell phone.

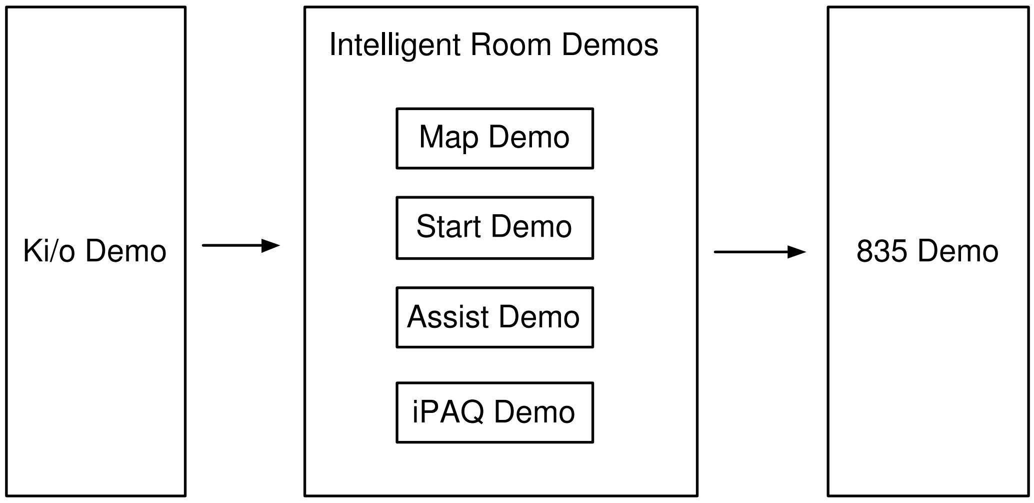


Figure 4-2: An example of a plan used by the Demo Manager is shown above. The demos in the Intelligent Room can be given in any order, and any of them can be skipped, at the discretion of the person giving the demo.

4.1.2 Building the *AIRE* Demo Manager

The Demo Manager uses plans to represent demos. One example of a demo plan is presented in Figure 4-2. In this demo, there are some parts that fall in a linear order; e.g., the Ki/o kiosk is the first project shown in the demo, the Intelligent Room follows, and the *AIRE*-space known as 835 concludes the demo. However, there is also a great deal of flexibility in the demo plan; the demos that take place in the Intelligent Room may be covered in any order, and some of them can be skipped at the discretion of the person giving the demo. Thus, the plan-based representation used by the Demo Manager allows it to support a degree of dynamism that distinguishes it from a simple macro recorder.

The components that make up the Demo Manager are shown in Figure 4-3. The Demo Manager was built using the Model-View-Controller design pattern. The model is the Planlet object that represents the demo plan and its current state. The view is the window that shows the demo's agenda and information about the current demo in the agenda. Updates to the model are made through the Demo Manager's plan monitor. The plan monitor makes updates to Planlet when notified by the demo

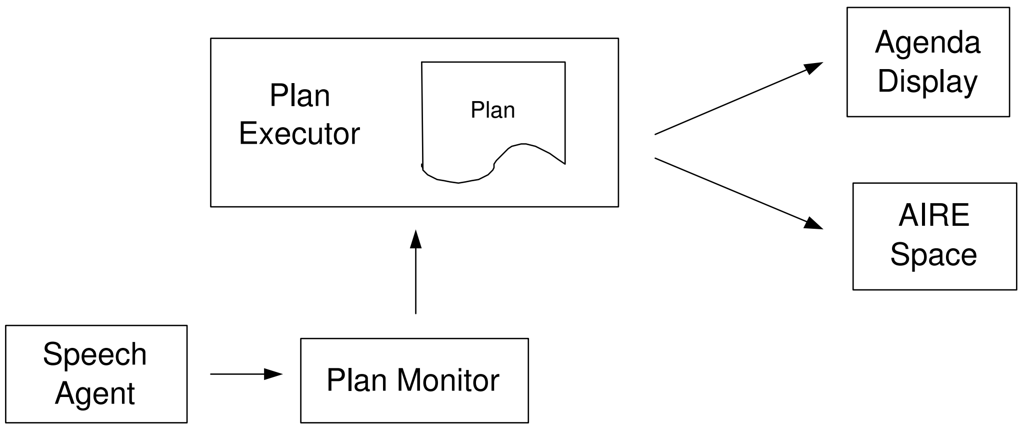


Figure 4-3: The components of the *AIRE* Demo Manager.

speech agent that the demo leader has completed different parts of the demo.

As described in Section 2.4, the executor interprets the annotations in each plan step and dispatches them when they are ready to execute. In the case of the Demo Manager, the annotations in a plan step include: who should perform the step (either the Demo Manager or the person leading the demo), what research project and which *AIRE*-space is being demonstrated (and the required resources to carry out that demo), and, if a person is introducing a particular demo, what speech utterances indicate he is ready to move on to the next demo. The speech agent uses this last annotation to set up the grammar it uses when listening to the demo leader.

4.2 The *AIRE*-space Configurator

The *AIRE* Demo Manager was designed for a fairly structured domain. Also, while it offers a certain degree of dynamism by allowing the demo leader to use the partial ordering of demos to vary his presentation, the possible projects to demonstrate are still specified in the plan. To see how a plan-based execution model would serve in less structured domains, we built the *AIRE*-space configurator to control the behavior of our *AIRE*-spaces so that they can better support people in their everyday tasks[3]. This system takes into account a user's current context and his preferences in those contexts when determining how to control the *AIRE*-space's behavior, thus providing

[3]This work is described in further detail elsewhere [41]

a larger degree of dynamism than the *AIRE* Demo Manager.

4.2.1 Scenarios

To illustrate how an *AIRE*-space can help a person in his everyday tasks, we present two simple, but very common scenarios that occur in an office environment and highlight what actions the *AIRE*-space configurator directs the *AIRE*-space to take.

Room Lighting

Howie's office can be lit in a number of ways. There are the fluorescent lights in the ceiling, a halogen torchiere in a corner, and sunlight from the window. In most situations, Howie prefers to work using the natural light from the window. However, there are some situations where this is not the case: in the winter months, afternoon work is best done by the Halogen lamp, whereas on warmer days when the air conditioning commonly breaks, the cooler light from the fluorescent lights is preferred to the added heat generated by the Halogen torchiere and the glare coming in from the window.

When Howie wants to make his office brighter or darker, he says aloud, "brighten" or "darken." Then, depending on the current context, his office either adjusts the lights and/or the drapes to achieve the desired effect.

Informal Meetings

Howie has scheduled a meeting with Gary to go over his quals talk. Since this is an important student meeting, five minutes before it starts, Howie's calendar agent reminds Howie of the meeting by sending a spoken notification to his office. Howie then asks his office to configure itself for Gary's practice talk. The office selects a plan to do this, and prepares to show the slides that Gary has emailed Howie earlier that day.

When Gary walks into Howie's office, the lights dim, the projector turns on and the Sony steerable camera turns on to record the talk. Gary gives his talk while Howie takes notes. After Gary is done, he sits down to listen to Howie's comments.

When this happens, the recording stops and the lights come on.

Of course, Howie holds other types of meetings in his office such as brainstorming sessions that require audio recording and white board capture, and confidential student meetings that require no recording of the meeting be made. His office configures itself for each of these situations when the need arises.

4.2.2 System Architecture

We take the view that a person's context influences his preferences, which in turn influences what plan he carries out and what resources he uses in those plans. For our *AIRE*-spaces to appropriately configure themselves with minimal user intervention, they need to be able to reason about these preferences and use them to select the plan that would configure the space to provide the highest utility to its occupants, given the current context.

Figure 4-4 presents an architectural overview of our system. The system consists of four main components: a plan selector that chooses which plan best achieves a user's goal, a resource manager that assigns resources to these plans, and a plan executor and a plan monitor that carry out the plan. As the plan is carried out, the plan executor may consult the resource manager since conditions may change while the plan is running, thus requiring a new resource assignment to the plan.

The plan executor and plan monitoring components are similar to what was discussed in Sections 2.4 and 4.1.2. The plan selector and resource management mechanisms are more advanced than the schemes used by the *AIRE* Demo Manager, however. In the case of the Demo Manager, specific resources are assigned to the demo, and the specific demo plan is selected by the person leading the demo. However, for the system to infer and accommodate the context-sensitive variations that come up in more informal situations, more complex schemes are required.

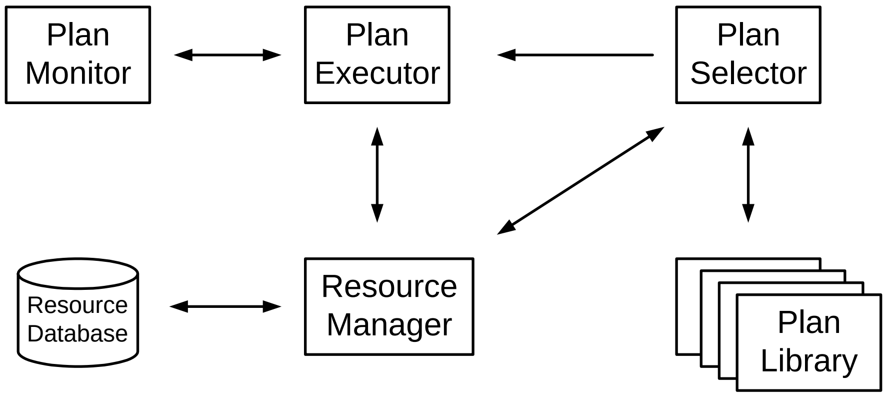


Figure 4-4: In our system, a plan selector chooses the best plan to achieve the user's goal from a plan library. The resource manager then assigns resources to this plan, using resources from its resource database. The resourced plan is then passed to the plan executor to be carried out. The plan monitor informs the plan executor as progress is made through the plan.

Plan Selection

The saying, "There are many ways to skin a cat" is illustrated in our first scenario, in which there are three different ways of lighting Howie's office. Each of these ways provides different qualities of service: for example, opening the drapes to let in sunlight produces a preferred lighting type, but impacts privacy. Exactly which of the three ways Howie prefers is influenced by what particular service qualities are important to Howie when he makes his request to brighten the room. An example of a statement of preferences between different service qualities (expressed in a lisp-like notation for readability) is given below. In words, it says that Howie prefers natural light twice as much as artificial light.

```
( (light-source natural)
  (>> 2)
  (light-source artificial) )
```

To select between multiple plans, the plan selector does a cost-benefit analysis between competing plans. The cost associated with a plan is the cost of using the resources required by that plan. The benefit is measured by determining how well that plan meets the service qualities specified by the requester's preferences. The plan

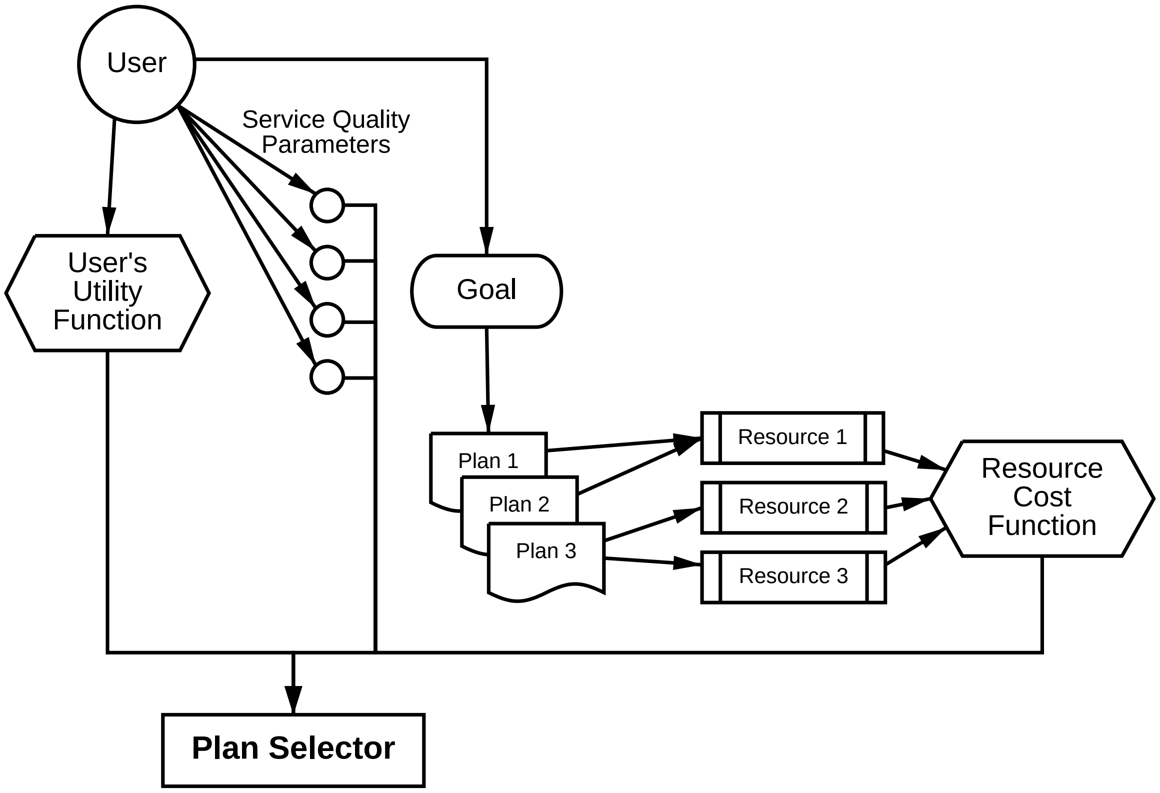


Figure 4-5: The plan selection mechanism. Figure courtesy of Stephen Peters.

with the highest benefit-to-cost ratio is chosen. The overall plan selection scheme is illustrated in Figure 4-5.

What sort of service quality a particular plan provides is listed as an annotation in that plan. For example, the plan that lights up Howie's office by opening the drapes would have the following annotations (the last annotation says that the intensity depends on how sunny it is outside): ``light-source=natural'', ``privacy=public'', and ``intensity=?amount-of-daylight''.

To compute how well a given plan meets a set of preferences, the plan selector uses techniques developed by McGeachie and Doyle [43]. In short, this method creates a directed acyclic graph where each node represents a possible setting to the service qualities. For example, let's say in our room lighting example there are three service qualities of note: `light-source`, `privacy`, and `intensity`. Then, there would be a node in the graph representing the setting `light-source=natural, privacy=public, intensity=high`. There is a directed edge from node N_1 to node

N_2 if the setting of service qualities represented by N_1 is preferred to those represented by N_2. The value of a node, and the benefit associated with the plan providing the quality of service represented by that node, is the length of the longest path originating from that node[4].

Resource Management

Similar to how a person's context influences his preferences over what service qualities are relevant in the plans under consideration, context also influences what resources are assigned to those plans. In the plans considered by the plan selector, resource requirements are described with generic resource descriptions. For example, in our lighting scenario, we have one plan that provides artificial light and requires a "lamp" resource and a second plan that provides natural light and requires a "drapes-controller" resource.

The resource manager searches for specific resources that meet the generic descriptions (e.g., either a halogen lamp or fluorescent lights fulfill the "lamp" resource requirement). To help fulfill resource requests, the resource manager uses a semantic network [52] to represent the relation between generic resource descriptions, specific resources and preferences between specific resources in different situations. For example, the semantic network can be used to represent the fact that when Howie needs to use a "lamp" resource, he prefers to use the halogen torchiere when he is reading and he prefers the fluorescent lights when he's working at his computer. This information is encoded into the semantic network using two separate entries, one for the halogen torchiere and the other for the fluorescent lights. When the resource manager has to find a "lamp" resource for Howie, it also takes any available context information and uses that to search the semantic network for the resource that is most appropriate for the current conditions.

[4]McGeachie and Doyle's paper [43] provides a fuller treatment of this topic, including a discussion of how to quickly construct the graph and how to handle preferences that introduce cycles into the graph.

4.3 Command Post for Autonomous Vehicles

Fleets of autonomous or remote-controlled vehicles have applications in a number of fields, such as space exploration and search-and-rescue. However, human supervision is still required to monitor the status of the fleet, to ensure that the mission is being carried out as planned. As the size of the fleet and the complexity of its mission increases, the difficulty of monitoring the mission status increases as well because human supervisors get overwhelmed with incoming data. In essence, it becomes difficult to "see the forest through the trees."

To make it easier for human supervisors to monitor the status of these missions, we created a Command Post application that presents an aggregated and abstracted view of the status information reported by the vehicles. This makes it easier for the mission commander to catch any possible threats to the successful completion of the mission.

4.3.1 Information Presented by the Command Post

The Command Post presents five different views of the mission's status. These views are:

- Vehicle View (see Figure 4-6). This view presents a live video feed from each of the autonomous vehicles taking part in the mission (i.e., the mission commander sees what the vehicle sees).

- God's Eye View (see Figure 4-7). This is a map giving the current location of all vehicles.

- Overall Plan View (see Figure 4-8). This view shows the mission plan the vehicles are carrying out, and their progress through that plan.

- Current Step View (see Figure 4-9). This table lists each vehicle and what step of the mission that vehicle is carrying out.

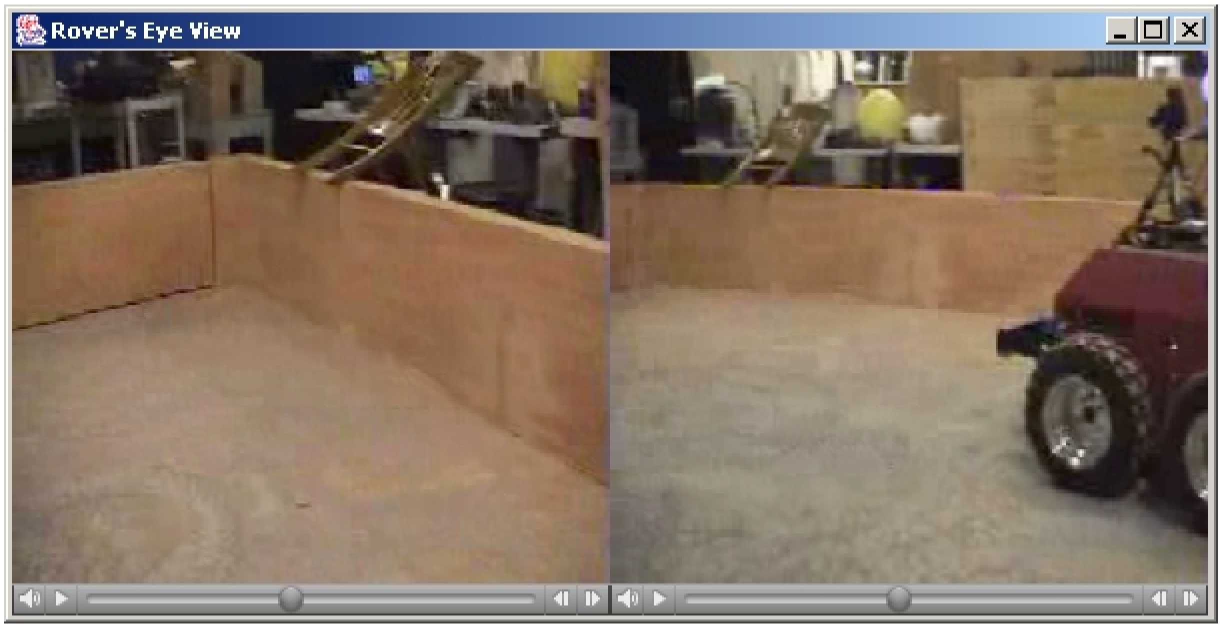


Figure 4-6: Vehicle View.

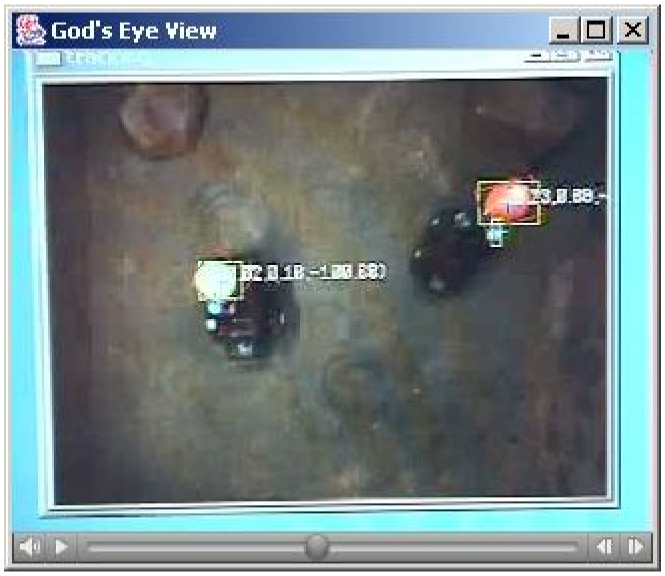


Figure 4-7: God's Eye View.

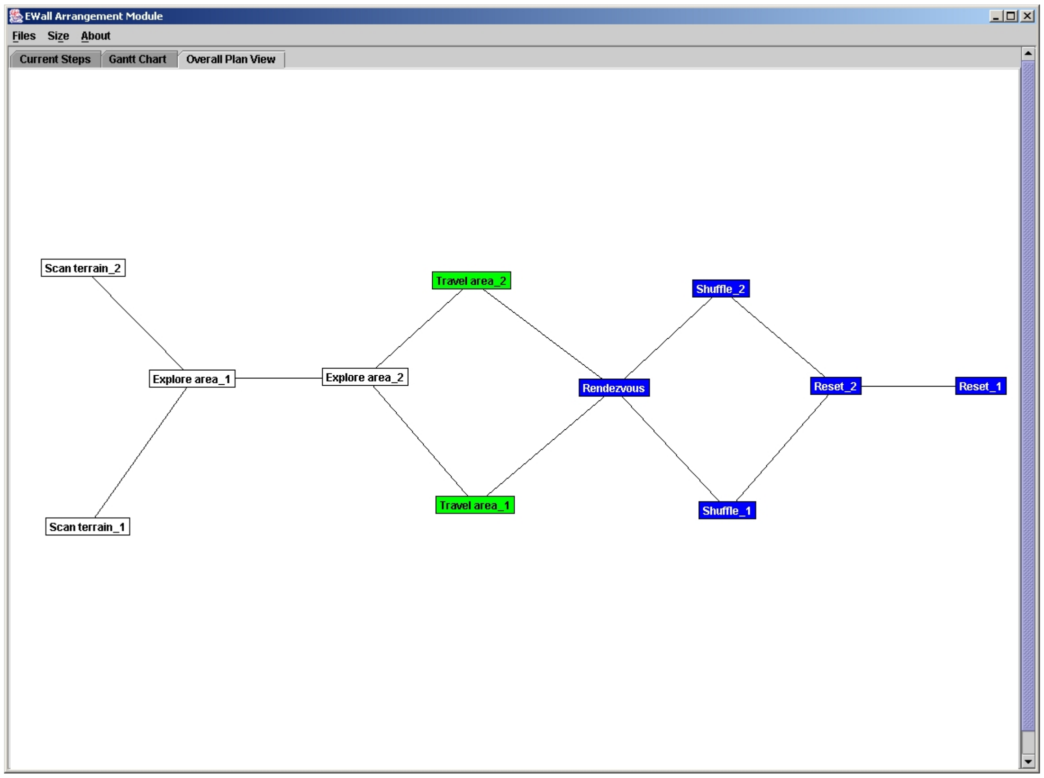
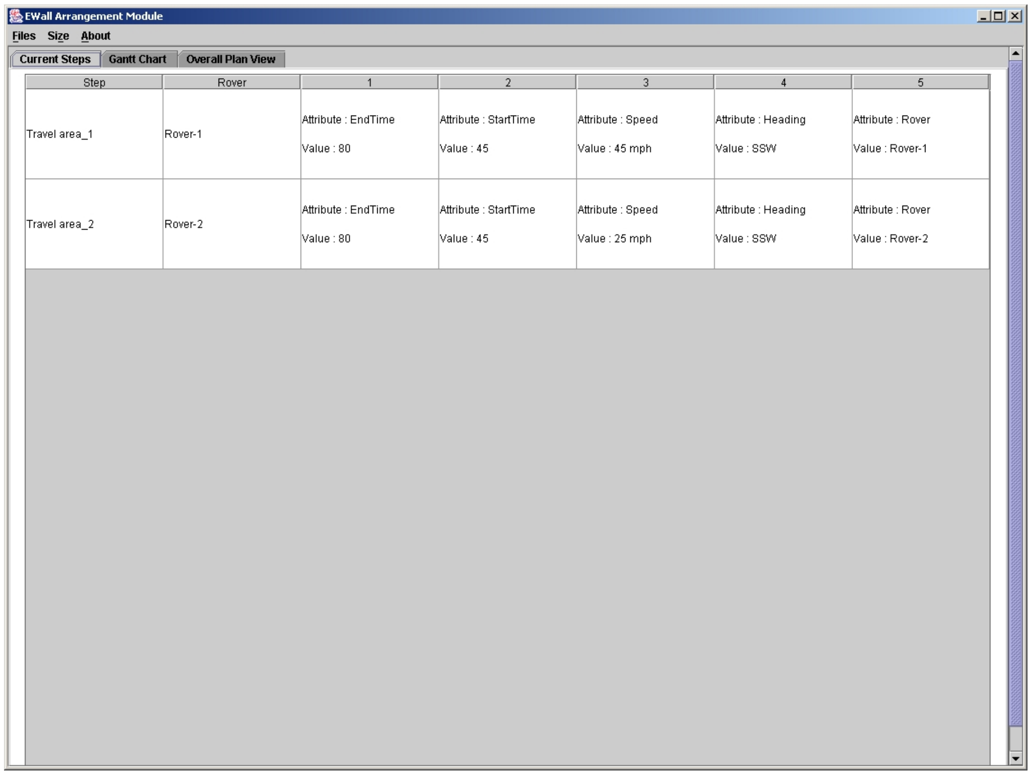


Figure 4-8: Overall Plan View. Steps are color coded to differentiate between those that have already been completed, those that are currently being carried out, and those that are waiting to have pre-requisites met.



Step	Rover	1	2	3	4	5
Travel area_1	Rover-1	Attribute : EndTime Value : 80	Attribute : StartTime Value : 45	Attribute : Speed Value : 45 mph	Attribute : Heading Value : SSW	Attribute : Rover Value : Rover-1
Travel area_2	Rover-2	Attribute : EndTime Value : 80	Attribute : StartTime Value : 45	Attribute : Speed Value : 25 mph	Attribute : Heading Value : SSW	Attribute : Rover Value : Rover-2

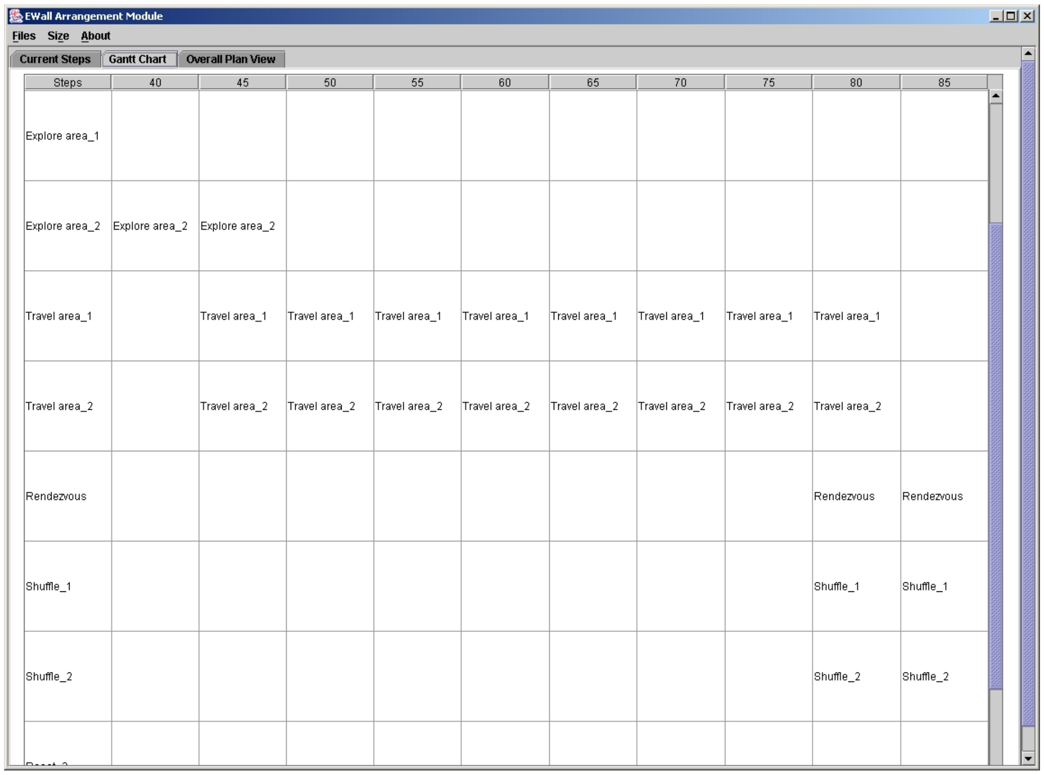
Figure 4-9: Current Step View.

- Gantt Chart (see Figure 4-10). This chart presents a timeline view of when steps in the mission need to be accomplished.

4.3.2 Command Post Architecture

Like the *AIRE* Demo Manager, the Command Post is built around the Model-View-Controller paradigm (see Figure 4-11). The Vehicle and God's Eye views draw from a streaming video source. The other three views represent different views of the Planlet plan and annotations used to represent the mission. Each of these views is based on Cardwall [11], a tool designed to facilitate easy visualization of related pieces of information.

Updates to the model are made by the executive that is coordinating the mission. In the scenarios that we have worked with, a model-based executive (see Section 7.2) controls the autonomous vehicles. This executive commands the fleet of vehicles using a plan representation that is mirrored by the Planlet plan that drives some



Steps	40	45	50	55	60	65	70	75	80	85
Explore area_1										
Explore area_2	Explore area_2	Explore area_2								
Travel area_1		Travel area_1	Travel area_1	Travel area_1	Travel area_1	Travel area_1	Travel area_1	Travel area_1	Travel area_1	
Travel area_2		Travel area_2	Travel area_2	Travel area_2	Travel area_2	Travel area_2	Travel area_2	Travel area_2	Travel area_2	
Rendezvous									Rendezvous	Rendezvous
Shuffle_1									Shuffle_1	Shuffle_1
Shuffle_2									Shuffle_2	Shuffle_2
[illegible]										

Figure 4-10: Gantt Chart.

of the Command Posts views. Since the Command Post is the newest plan-based applications we have developed, work is currently underway to integrate notifications from the model-based executive with the API the Command Post exposes to update the Planlet model.

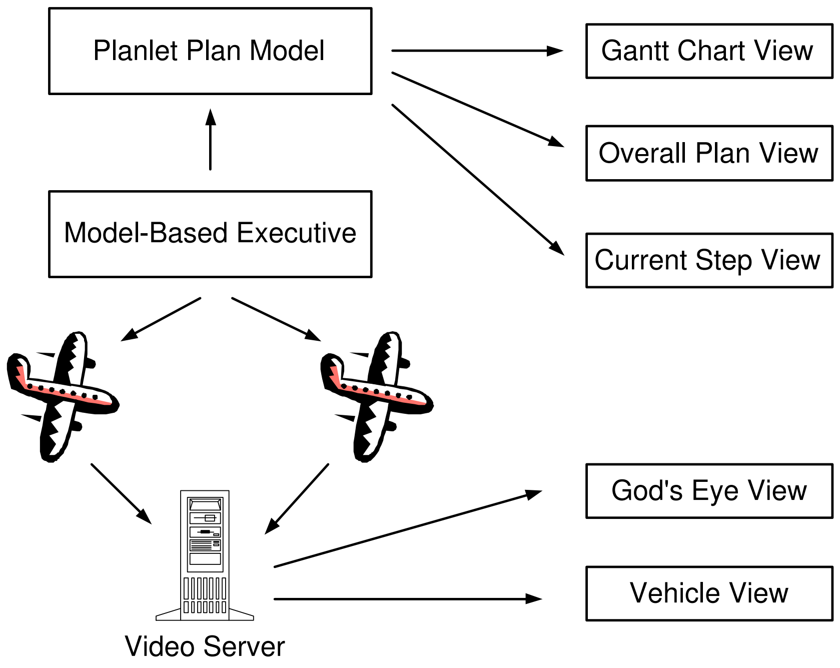


Figure 4-11: A model-based executive controls a fleet of autonomous vehicles. Three of the views presented by the Command Post are based on the Planlet representation of the plan this executive is carrying out. The other two views show video footage and location information provided by each of the vehicles.

Chapter 5

Evaluation

Section 1.5 lists a number of benefits that plan-based proactive computing provides. For convenience, we list them again below, in abbreviated form:

- A higher level of abstraction for designing ubicomp applications.
- A higher level of abstraction for end users to communicate intent.
- Automatic configuration of environments.
- Better informed systems.
- Self-adaptive systems.

We revisit these benefits in this chapter, and see how the applications described in Chapter 4 demonstrate these benefits.

5.1 Plans as an Abstraction for Programmers

The applications we built using a plan-driven programming paradigm — the *AIRE* Demo Manager, the *AIRE*-space configurator, and the Command Post — represent a wide range of domains. Giving a demo is a very structured task in which the person leading the demo can be regarded as following a script. Configuring a space to support a particular activity can be done in a well-defined sequence of steps, but the actual activity that takes place in that space can be very unconstrained and free-flowing. Finally, supervising the progress of a mission requires a larger degree of focus on processing a wealth of information than in the first two domains. Yet, despite the different requirements found in each of these domains, the same plan-driven programming paradigm was used to create useful applications for all of them.

In addition to being a useful programming paradigm for different application domains, using plans to guide the behavior of applications allows application developers to decouple the design of an application's behavior from the actual implementation of that behavior. The obvious benefit of this decoupling is that it allows developers to focus separately on these two issues. For example, with the *AIRE*-space configurator, the application developer only needed to specify *what* things the *AIRE*-space needs to do in order to configure itself per the user's request. The actual decision-making regarding *how* this is to be done is left to other components, in this case, the plan selector and resource manager.

Also, using plans as the basis for program execution allows developers to choose from a number of different plan sources, ranging from hierarchical task network planners [24] to human plan editors. The ability to choose where plans come from allows a developer to either pass off the responsibility of generating plans to the well-established field of AI planning [67] or assume that responsibility himself if human decision-making is required in the planning process.

When developers choose to write their own plans, it is important that they have an easy and straight-forward way of doing this. Early versions of Planlet did not allow Planlet plans to be created from an XML descriptions. This became a major inconvenience because the programmatic interface would require recompilation of code for each new plan. When the XML interface was added, creating new plans became a simple matter of editing an XML file, thus making the plan definition process much easier.

Finally, plans provide a nice modular way of encapsulating behavior. While this has more of an impact for the end-user (see Section 5.2), this also benefits the application developer in that plans can be reused in other plans. Also, plans provide a natural way of grouping resource needs, and this information can be passed to a resource manager for use in determining the optimal resource allocations to the plan.

5.2 Plans as an Abstraction for End Users

From the perspective of an end-user, plans capture the structure of a task and present it at an appropriate level of abstraction to the user. The three plan-based applications described in this thesis each show different aspects of how this representation can be used to support how a user approaches his tasks and how he perceives events that occur during the course of that task.

Since our plan representation uses a partial-ordering of the plan steps, our plan-based applications are able to identify all possible steps that a person may take next and use the corresponding annotations in those steps to prepare for any of them. While plans do impose some ordering on their steps, depending on context and a person's preferences, the total ordering of a plan's steps will vary when a person actually carries out his plan. This property is illustrated in the *AIRE* Demo Manager when there are multiple items that could be demoed next. The Demo Manager just informs the person leading the demo of his options, and then the demo leader uses his discretion to choose which items he wants to show off. This ability to reason about what a user could do next, informing him of these options, and then proactively preparing for them is a feature applications that artificially impose a total ordering on a task or are structured around a set of event handlers cannot provide.

Just as a person's context and preferences will influence the order in which he carries out tasks, they will also influence how he goes about performing a given task and how he would want a particular task to be carried out [9]. Because of this, it is necessary to support a late-binding to the actual implementation of the steps described in a plan. This capability is what distinguishes an application like the *AIRE*-space configurator from traditional workflow systems. In traditional workflow systems, the task structure is static, and it rigidly dictates what a person should do. The plan-driven execution model used in the *AIRE*-space configurator, however, allows a much greater degree of dynamism at plan execution time by taking into account a user's context, expressed as his current preferences, and by using a knowledge-based resource management system.

Finally, plans define the level of detail a person is concerned with. Applications can use this fact to structure how information can be best presented. This is most clearly demonstrated in the Command Post application. Instead of focusing on the individual movements of each of the vehicles in the fleet, the mission commander is able to see the 10,000 foot view of the mission status using the overall plan view and the Gantt chart view. If the commander wants more specific information regarding an individual unit, he can look at the view that describes what part of the mission each unit is currently carrying out. All of these views are facilitated using the underlying plan representation of the mission.

5.3 Automatic Configuration of Environments

One of the first benefits envisioned from plan-based applications was the ability to proactively configure devices or spaces a person may soon use or enter. As this thesis has developed, this feature has become a canonical part of many of the scenarios illustrating what a plan-based application can do. The reason for this is that proactively configuring devices and spaces is relatively simple, especially when given knowledge of what a person is and will soon be doing. Yet, this goes a long way in reducing user distraction by allowing a person to smoothly transition to the next step in his plan while leaving the task of setting up the devices and environment to his plan-based application.

5.4 Better Informed Systems

Plans provide applications with an inherent sense of user purpose and knowledge about a user's context. This information can be used to proactively assist a person, as described in Section 5.3, but there are other benefits as well. After the applications in this thesis were written, it became clear that merely making the user's plan visible to him, provided value, even if the application did not proactively act upon the information.

The benefits that plan visibility provide are clearly seen in the Command Post application, where the whole intent of the application is to provide a mission commander with the appropriate information to keep him updated on the mission's progress. The commander can use both the Gantt chart view and the overall plan view to assess, at a glance, the status of the mission, thus helping him further focus and direct his attention to a particular aspect of the mission. The *AIRE* Demo Manager is another example where plan visibility benefits the application's users. The agenda and information about the current demo help the demo leader structure his presentation. Since the visible agenda provides a record of what has, is, and will be demoed, it also helps maintain a sense of continuity for the people whom the demo is being shown to.

5.5 Self-adaptive Systems

The fifth benefit provided by plan-based applications is that their plans provide a ready basis for use in model-based diagnosis and recovery techniques [20, 60]. Although the work described in this thesis does not incorporate these techniques to recover from system failure, early work is underway in this direction. Specifically, each plan is converted to a Bayes net that is then used to perform inference to identify which step in the plan is most likely to have failed.

5.6 When Plans Are Appropriate

A plan-based programming paradigm is best suited for domains that have some sort of structure. The application domains highlighted in this thesis exhibit this property. However, in domains without standard protocols, where it is difficult to model a person's behavior, this paradigm is not very effective. For example, on a relaxed

Saturday afternoon, when a person has the entire afternoon in front of him, what should his *AIRE*-space configurator do? Clearly, forcing a plan for the sake of the plan is not the right approach; indeed, in situations like this, it is best to be in a reactive as opposed to proactive mode.

Chapter 6

Future Work

The work done in this thesis has suggested a number of areas for future work. The first is adding more functionality to the Planlet code base. The second is increasing the perceptual and reasoning abilities of plan-based applications so that they can better correlate observations with the completion of certain plan steps. A third area is making improvements to the plan-based applications described in this thesis.

6.1 The Evolving Planlet

There are a number of functionalities that could be added to the Planlet code base to make it easier to use and more useful when building plan-based applications. The first functionality is a graphical plan editor that could be used to create plans. This would nicely complement the programmatic and XML interfaces that Planlet currently provides. Initial steps in this direction will use Cardwall [11] as the plan editing interface to manipulate the underlying Planlet data structures.

A second plan editing functionality that would be useful and fairly straightforward to implement would be the ability to save, in an XML file, the data and state Planlet plans acquire as they are carried out. This would allow plans to be checkpointed and resumed at a later time and would also serve as a record of the plan's execution. This could be implemented using the same third-party Java-XML toolkit [36] used to translate XML plan descriptions into Planlet objects; another alternative would be to use Java's XML encoder to do the task [35].

Expanding the types of annotations that Planlet supports is another possible feature that can be added. As mentioned in Section 2.2.1, Planlet currently supports string and numeric type annotations, in the sense that annotations of these types can be evaluated in terms of other annotations. Other types of annotations can be stored and later referenced in the Planlet data structure, but the current version of Planlet does not support any sense of evaluating these types of annotations.

6.2 Reasoning About Progress Through Plans

There are also many possibilities for future work relating to how Planlet and plan-based applications reason about a user's progress as he goes through his plan steps. For example, Planlet currently views all next possible steps as equally likely events. However, just because an event could possibly happen next does not mean it is a plausible next step. Where the user is currently located, the current time of day, and his preference to doing certain plan steps over others all influence what a user is expected to do next. A future version of Planlet could take context into account to generate a probabilistic model of what steps the user is most likely to do next, and the plan executor could act on this additional piece of knowledge.

Allowing a plan-based application to determine its own proactive intervention strategy is another area of future work. While applications like the *AIRE*-space configurator make use of Planlet's ability to late-bind the description of a plan step to the actual implementation of that step, the plan writer still has to specify the fact that he wants this behavior. From a plan writer's perspective, it would be easier to think solely from a person's point of view, and describe only the steps that person is concerned with. Advanced plan-based applications could then take this description and its associated annotations and make inferences as to how it could support the person at each step, as opposed to consulting the plan for that information.

One way of allowing applications to determine their own intervention strategy would be to adopt a plug-in architecture similar to Collagen's [57]. Collagen is a system that uses collaborative discourse theory to guide a software agent's interaction with a user and the application he is working with[1]. Collagen plug-ins are essentially rules that provide a scored response describing what the software agent should do under a particular set of circumstances. Since many plug-ins may suggest differing responses under the same set of circumstances, the Collagen agent chooses the response with the highest score.

The dual of having an application proactively assist a user is having the application recover from times when it mistakes what step a person is working on and then performs some action that actually distracts the person from his actual task. In some situations, it may not be possible to discreetly "unring the bell," such as when the *AIRE* Demo Manager misinterprets what the demo leader just said and mistakenly starts another demo and updates the agenda. In cases like this, a person may have to manually nudge the application back on track. However, there are other times when it is possible for an application to automatically recover from misinterpreting a user's actions. For example, the *AIRE*-space configurator may prematurely activate devices in an *AIRE*-space that won't be visited in the near future. Recovery in this case is simple: turn the devices off. Distinguishing between these two situations and reacting appropriately is a challenging problem.

6.3 Perceiving Progress Through Plans

Improving the mechanism the plan monitoring component uses to determine that a person has completed a step is another area of future work. One way of doing this is to use some sort of avatar to represent the application [12]. The user would then communicate directly with the avatar to explicitly indicate that he has finished a particular step. The *AIRE* group has such an avatar named SAM. SAM has been used in conjunction with a head-pose tracker developed by the Vision Interface Group [64] to create a "look-to-talk" (LTT) interface [48] for directing spoken utterances to a software agent (see Figure 6-1). A person can use LTT to explicitly notify an

[1]See Section 7.4 for more details about Collagen.

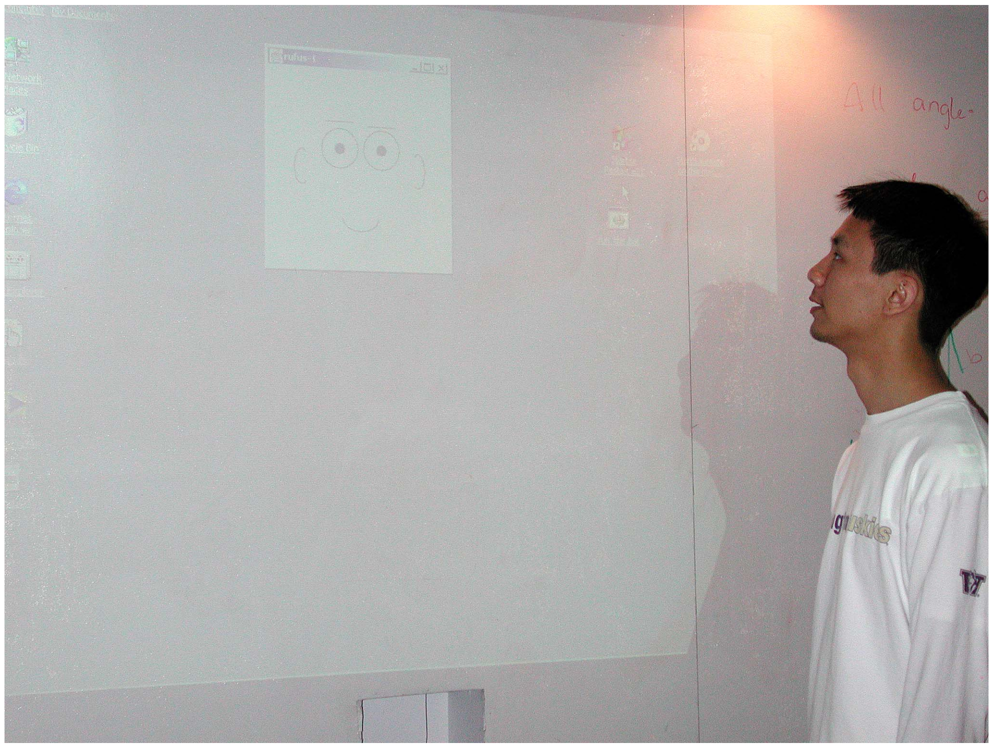


Figure 6-1: A person interacting with SAM.

application when he has finished a particular task.

In addition to the explicit cues provided by the user to indicate that he is done with a step, future implementations of the plan monitor could infer from perceptual cues what activities are occurring and when they complete. Current research in the *AIRE* group uses stereo cameras and a blackboard [46, 47] to learn how people use different regions of a physical space [21]. The resulting activity zones that are identified can then be used along with a person's movements between zones to infer when one step in a plan completes and the next step begins. By using activity zones in this manner, we would combine the bottom-up data provided by the activity zones with the top-down structure provided by plans.

6.4 Improving the Existing Applications

The current suite of applications described in this thesis illustrate the benefits of a plan-based programming paradigm. However, there are a number of things that can be done to improve their usefulness and further demonstrate the advantages of the paradigm.

One feature to add to the *AIRE* Demo Manager would be the ability to dynamically adjust the demo's agenda as the demo progressed. With this capability, not only would the Demo Manager be able to warn a demo leader if he was running short on time, it would also be able to drop demos that haven't been shown and/or reduce the amount of displayed information that the demo leader would speak about.

Also, as suggested in Section 6.3, activity zones could be used to indicate when a person has finished demoing a particular *AIRE*-space. The use of activity zones could also be used by the *AIRE*-space configurator to determine when to change the setup of the space. For example, in the quals talk scenario in Section 4.2.1, activity zones could be used to infer when Gary is finished with his practice talk.

Lastly, there are two main areas of future work with regards to the Command Post application. The first piece of work was mentioned in Section 4.3.2: integrating the Command Post with the actual executor that controls the fleet of autonomous vehicles. Although there is an API for updating the different views presented by the Command Post, it is not currently being used by the executor. The second piece of work would be to add a knowledge component that could use the annotations in each plan step to highlight which steps a mission commander should focus on. Although the Command Post's plan view and Gantt chart view provide some minimal guidance as to which step to focus on, the most critical step in the mission may not be immediately apparent. Using the plan annotations to actively help direct the mission commander's attention could help him identify potential problems he may otherwise miss.

Chapter 7

Related Work

While there are many research projects in the area of plan-directed systems and in the area of ubiquitous computing, little work has been done in the overlap of the two areas, which is the focus of this thesis. As the field of ubiquitous computing has matured, however, researchers are starting to realize that issues such as planning, plan execution, and plan monitoring, concepts traditionally associated with Artificial Intelligence, should be considered when creating a robust ubicomp application. This chapter examines what work has been done to incorporate the concept of plans into ubicomp applications, how plans are used to direct the behavior of other autonomous systems, and other approaches to directing the behavior of ubicomp applications.

7.1 Task-Driven Computing

As mentioned in Section 1.5, although the high-level notion of a task has been introduced to ubiquitous computing, the formal notion of a plan is still missing. Wang [65] first introduces the notion of task-driven computing as a means of directing how the Aura infrastructure [29] should configure devices so that the user can maintain a sense of continuity as he moves from one place to another. In this model, tasks are viewed as a collection of services, and the service descriptions are used to find appropriate resources that provide those services. The emphasis in this model is supporting a mobile user as he works on a general task that can be done with many applications (editing a paper or watching a video are the canonical examples). To this end, a service-oriented view of a task is sufficient.

However, this notion of a task is somewhat limited to officework scenarios where providing the required services is all that could be done to assist a user. In other domains, tasks have a far richer connotation that includes a description of specific steps that must be done and dependencies between steps. In these domains, ubicomp applications have a far greater potential to assist users by actually performing the mundane plan steps or warning a user if he is doing something that would run counter to his task. Building ubicomp applications using a plan-driven execution model allows for added user benefit in these domains whereas a service-oriented view of tasks does not.

Christensen furthers the idea of a task-driven computing model by describing an early prototype of an activity-centered computing infrastructure to support hospital workers [16]. Similar to Aura, in this system, tasks are treated as first-class objects that serve as abstractions for applications, their state, and the data related to the activity. The most novel aspect of this system is a rule-based expert system that infers likely user activity based on context. Initial experiences with this system have been promising, but also raise many issues. Among these issues are how can the system provide additional value to its users. Currently the system provides support for "follow-me" applications (mainly data access) but does not support non-composite activities. A plan-based model of activities would provide insight as to how additional support could be provided, and would make it easier to compose simpler activities into more complex ones.

Finally, in the Open Agent Architecture (OAA) [42], there is a facilitator agent that coordinates interaction between agents. In OAA, the facilitator agent performs task management by recursively decomposing a task into subgoals in a PROLOG-like, backward-chaining manner. The facilitator then routes subgoals to the agents that can achieve them. This approach is very different from a plan-driven execution model in a number of ways. While the requesting agent's intent is encapsulated in the goal, the resolution mechanism and actual means by which the goal is achieved is abstracted away from the requester. In the plan-driven execution model presented in this thesis, the person who wants to achieve a goal is an active player in carrying out the actions that achieve his goal, and therefore OAA's task management approach is similar to the other "task-as-collection-of-services" models presented in this section.

7.2 Model-Based Programming

An example of a plan-based system is the work that has been done in model-based programming [68, 69]. This work is concerned with building robust, autonomous space vehicles. The approach taken in this work is to provide the designers of these vehicles a programming language that "raises the bar" in terms of the level of abstraction used to program the vehicles. Namely, model-based programming introduces a programming paradigm that allows programmers to think in terms of states and transitions between these states, as opposed to the lower level (and more error-prone) embedded programming of individual devices.

Once the model-based program is written, it is then given to a model-based executive that translates the program into a set of goals and plans that would achieve those goals. These plans are then carried out by the executive. As an example, if the model-based program sets a rocket engine to the "thrusting" state, the model-based executive would verify that the engine is in the "off" state and that it was functional. It would then plan the necessary actions to open a series of valves to achieve thrust. If the executive determines that one of the valves needed to achieve thrust in the original plan is in a failure mode, then the executive would plan an alternate course of action and carry that out.

Model-based programming has been very effective in programming autonomous space vehicles. However, the domain of ubiquitous computing poses other demands on the plan representation used to represent the task. In particular, it is necessary to consider the human-in-the-loop factor; in the case of ubiquitous computing, the infrastructure should *assist* the user with *his* plan of action, whereas in the case of autonomous vehicles, the vehicle is carrying out its own plan. Since the user is the one who is ultimately being served, depending on its domain, plan-based ubicomp applications may need a more accommodating enforcement policy with regards to how closely a user follows his plan.

7.3 Cognitive Orthotics

An emerging domain for plan monitoring systems is cognitive orthotics. This field is concerned with developing tools to help elderly people with the activities of daily life (e.g., personal hygiene, recreation, taking medication, etc.) and thereby allow them to maintain their independence as they age.

Autominder [55] is a plan monitoring agent that keeps track of the different daily activities an elderly person is expected to do and offers appropriate reminders when a person forgets to do something he should do. Unlike the deterministic model of plans presented in this thesis, Autominder represents plans as a Disjunctive Temporal Problem (DTP). Under this model, a user's plan is regarded as a set of events and a set of disjunctive temporal constraints between events. As a person performs different actions, Autominder makes sure that the DTP representing the user's plan is consistent by sending reminders to perform certain acts if one of the DTP's constraints may soon be violated (e.g., take medication one hour before eating). To quickly determine if a DTP is consistent, Autominder treats the DTP as a constraint satisfaction problem.

The time at which reminders are made also takes into account the user's expected behavior, which is modeled using a Quantitative Temporal Dynamic Bayes Net[1]. By taking into account both a user's expected behavior and the necessity of the reminder, Autominder can justify why reminders are made and provide useful reminders without making its user dependent on the system. Autominder is part of the software that runs on a personal robotic assistant. This robot performs the sensor fusion to infer user actions (so Autominder receives notifications of the form, "the client entered the

[1]A Quantitative Temporal Dynamic Bayes Net combines a standard Bayes net with a dynamic Bayes net to reason about both the temporal aspects of the person's activities (using the standard Bayes net) and the activities the person is currently doing (using the dynamic Bayes net)

bathroom") and is the one that actually reminds the person to do certain things.

The main difference between our work and the work being done in cognitive orthotics is the primary beneficiaries. In our work, we are interested in providing assistance to people who are working on fairly structured tasks, and so our representation consists of discrete steps and causal links between those steps. Cognitive orthotics, on the other hand, is interested in assisting elderly people maintain their independence in the less structured activities of daily life. For this reason, a schedule-based model that takes into account preferences is more appropriate for this domain.

7.4 SharedPlans for Collaboration

The SharedPlans theory for collaboration [49] is a linguistic model of dialog structure. In addition to the concept of plan steps and causal links between steps, the model also includes reasoning about group intention, the group's current focus of attention, and what individuals believe about their own abilities and the abilities of others to contribute to the group intention. These added concepts allow for an incremental refinement of the actual steps carried out in a plan, essentially modeling the nature of collaboration. We mention this idea here because computer science has applied this model to facilitate better human-computer interaction.

Collagen [58] is a direct implementation of the SharedPlans theory. Applications written with Collagen use a software agent to collaborate with a user as he works on his task. The agent maintains a model of what the user is trying to do by having a dialog with the user (this dialog includes recording user activities and asking for clarification when necessary). As the user completes parts of his task (and the dialog between user and agent continues) the agent works with the user to perform certain actions when this interaction would be useful. The resulting interaction is similar to two people working together in front of a whiteboard to solve a problem. Collagen has been used to create agents that help users with the following tasks: program their VCRs, set their thermostats, and guide students through the use of simulators.

DIAL [49] is another system based on SharedPlans. DIAL is a web-based interface used to help students retrieve relevant course material. By using the SharedPlan model and exposing its model of the current discourse state, DIAL is able to maintain and update a shared sense of context with the user, thus reducing the amount of explicit information a student would need to provide in his query and increasing the signal-to-noise ratio in the information DIAL retrieves. For example, DIAL is able to distinguish between the following requests: a request from a student who is studying for an exam by searching for material from the lecture on "Programming with loops" and a request from a student who first started studying for the exam but then changed his mind and decided to review the lecture on loops (with the intent at reviewing the entire lecture, not just the material relevant for the exam).

Collagen's and DIAL's use of SharedPlans is very similar to the plan-based execution model presented in this thesis. However, the applications developed using this theory have revolved around desktop applications where the computer can be regarded as a peer to the user, and thus there is a definite sense of collaborative human-computer interaction. This thesis, on the other hand, is concerned with ubiquitous computing domains where the user has a fairly defined sense of what he wants to do, and the tasks involved. The application, therefore, needs to account for resource and infrastructure concerns while it plays more of a supporting, as opposed to collaborative, role.

7.5 Alternate Approaches to Proactive Computing

Antifakos describes how plans can be used to proactively assist people assembling furniture [6]. In this system, the plan for furniture assembly is represented as a finite state machine in which nodes represent world states and transitions between nodes represent user actions. While the paper describes an impressive prototype, the focus is more on using Hidden Markov Models to infer when transitions between world states have been completed. This is a very important capability to have, but it is not extensively used in conjunction with the plan representation to achieve proactivity (for example, although proposed, it is not clear how the system would warn a user that he has incorrectly oriented a piece of particle board in the assembly). In the plan-based execution model presented in this thesis, exploiting knowledge of user plans for the sake of proactive assistance is one of the driving motivations.

Lumiere (the Microsoft research project that was commercialized into the Office Assistant, aka “Clippy”) uses Bayesian user modeling to guide Microsoft Office’s proactive editing measures [34]. Instead of using a plan representation of a user’s task, Lumiere uses dynamic Bayes nets to model the dependencies between a user’s expertise, the current state of the Office application, and the history of the user’s interaction with the application. This model is then used in conjunction with a utility function to infer what the user is currently working on and whether or not it would be beneficial to the user if the application were to offer some sort of assistance (requests to help the user write a letter not withstanding).

7.6 Subsumption Architectures

ReBa [39] is a system designed to control how MIT’s Intelligent Room reacts to user activities in different circumstances. Reactions to particular actions are defined in a behavior. Behaviors can be layered over one another with the behavior on top overriding the lower behavior when two or more behaviors conflict. For example, the Intelligent Room could have an “individuals working” behavior that sends any important pages to the room’s scrolling marquee and lowers the sound coming from the main speakers if another person enters the room. A separate “movie presentation” behavior that prevents the main speakers from lowering their volume could be layered over the “individuals working” behavior. Then, if during the working session, the people in the room need to review a particular movie clip, any important pages will still be routed to the scrolling marquee, but the movie will not be interrupted when a new person enters the room.

Our approach is different than the one used in ReBa in that we use plans to explicitly model user activities. Individual events, therefore, are not isolated occurrences. Rather, they have a larger context against which the application can leverage. For example, by knowing where a particular step fits into a larger plan, applications can explain why a certain step should be performed. Applications can also take actions in anticipation of what the user will do next, such as reserve resources that may be used by the user in the near future, rather than wait for the user to do something before the application reacts to it.

Chapter 8

Contributions

Advances in ubiquitous computing have allowed programmers to build applications that use a myriad of devices to help people in their everyday tasks. While this research has made such applications possible, the question of how these applications *should* make use of the devices available to them has not received much attention. To address this matter, this thesis presented a plan-based programming paradigm, based on standard principles from AI planning and plan monitoring, for creating proactive ubicomp applications.

Three applications, each from different domains, were built using this paradigm. A number of lessons were learned from this experience. The two most significant ones are that plans provide a powerful level of abstraction for programmers and that having applications present information and regard events at the level of plan steps is a useful abstraction from the end-user's perspective.

Creating additional layers of functionality on top of the core Planlet plan layer is an area of much future work. In particular, formalizing and implementing protocols that use a plan's annotations to infer what proactive measures an application should take and when a particular plan step has completed are interesting future research directions.

Appendix A

Sample XML Description of a Planlet Plan

```
<?xml version='1.0' encoding='utf-8'?>
<hierarchicalplan>
  <name> Setup devices for practice talk </name>
  <numinputs> 0 </numinputs>
  <numoutputs> 1 </numoutputs>

  <subplans>

    <step>
      <name> Start Recording Video </name>
      <numinputs> 0 </numinputs>
      <numoutputs> 1 </numoutputs>
      <properties>
        <property name="resource" type="String" value="video camera"/>
        <property name="method" type="String" value="record"/>
      </properties>
    </step>

    <step>
    <name> Dim Lights </name>
    <numinputs> 0 </numinputs>
    <numoutputs> 1 </numoutputs>
    <properties>
      <property name="resource" type="String" value="lamp"/>
      <property name="method" type="String" value="dim"/>
    </properties>
  </step>

  <step>
    <name> Start Presentation Software </name>
    <numinputs> 0 </numinputs>
    <numoutputs> 1 </numoutputs>
    <properties>
      <property name="resource" type="String" value="powerpoint"/>
      <property name="method" type="String" value="show"/>
      <property name="file" type="String" value="talk.ppt"/>
    </properties>
  </step>

  <step>
    <name> Turn On Projector </name>
    <numinputs> 3 </numinputs>
    <numoutputs> 1 </numoutputs>
    <properties>
      <property name="resource" type="String" value="projector"/>
      <property name="method" type="String" value="on"/>
    </properties>
  </step>

</subplans>
```

```
<flows>

  <flow>
    <producerplanname> Dim Lights </producerplanname>
    <producerinouttype> output </producerinouttype>
    <producerportnum> 0 </producerportnum>
    <consumerplanname> Turn on projector </consumerplanname>
    <consumerinouttype> input </consumerinouttype>
    <consumerportnum> 0 </consumerportnum>
  </flow>

  <flow>
    <producerplanname> Start Presentation Software </producerplanname>
    <producerinouttype> output </producerinouttype>
    <producerportnum> 0 </producerportnum>
    <consumerplanname> Turn On Projector </consumerplanname>
    <consumerinouttype> input </consumerinouttype>
    <consumerportnum> 1 </consumerportnum>
  </flow>


  <flow>
    <producerplanname> Start Recording Video </producerplanname>
    <producerinouttype> output </producerinouttype>
    <producerportnum> 0 </producerportnum>
    <consumerplanname> Turn On Projector </consumerplanname>
    <consumerinouttype> input </consumerinouttype>
    <consumerportnum> 2 </consumerportnum>
  </flow>

</flows>
```

```
</hierarchicalplan>
```

Bibliography

[1] Greg Abowd, Christopher Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3, 1997.

[2] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th SOSP*, pages 186–201, December 1999.

[3] Aaron Adler. Segmentation and Alignment of Speech and Sketching in a Design Environment. Masters of Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

[4] *AIRE* home page. http://www.ai.mit.edu/projects/aire/.

[5] J.F. Allen, H.A. Kautz, R.N. Pelavin, and J.D. Tennenberg, editors. *Reasoning About Plans*, chapter 2, pages 69–126. Morgan Kaufmann Publishers, 1991.

[6] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. Proactive instructions for furniture assembly. In *Proceedings of UbiComp 2002*, pages 351–360, Göteberg, Sweden, 2002.

[7] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Reading, MA, 1999.

[8] Larry Arnstein, Gaetano Borriello, Sunny Consolvo, Chia-Yang Hung, and Jing Su. Labscape: A smart environment for the cell biology laboratory. *IEEE Pervasive Computing Magazine*, 1(3):13–21, July–September 2002.

[9] Jakob Bardram. Plans as situated action: An activity theory approach to workflow systems. In *Proceedings of the Fifth European Conference on Computer Supported CooperativeWork (ECSCW)*, Lancaster, U.K., 1997.

[10] Richard A. Bolt. Put-That-There: Voice and gesture at the graphics interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Technologies*, pages 262–270, 1980.

[11] Cardwall home page. http://ewall.mit.edu/.

[12] Justine Cassell, Timothy W. Bickmore, Mark Billinghurst, L. Campbell, K. Chang, Hannes Hogni Vilhjalmsson, and H. Yan. Embodiment in conversational interfaces: Rea. In *CHI*, pages 520–527, 1999.

[13] Andy Chang. AFAIK: A help system for the Intelligent Room. Masters of Engineering Thesis, Massachusetts Institute of Technology, October 2001.

[14] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a context-aware electronic tourist guide: Some issues and experiences. In *Proceedings of CHI 2000*, pages 17–24, April 2000.

[15] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Experiences of developing and deploying a context-aware tourist guide: The guide project. In *Proceedings of MobiCom 2000*, pages 20–31, August 2000.

[16] Henrik Bærbak Christensen and Jakob E. Bardram. Supporting human activities — exploring activity-centered computing. In *Proceedings of UbiComp 2002*, pages 107–116, Göteberg, Sweden, 2002.

[17] Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *1st International Workshop on Managing Interactions in Smart Environments (MANSE'99)*, pages 201–212, Dublin, Ireland, December 1999. Springer-Verlag.

[18] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3D system for novices. In *Proceedings of CHI 2000*, pages 486–493, 2000.

[19] Steven Czerwinski, Ben Zhao, Todd Hodges, Anthony Joseph, and Randy Katz. An Architecture for a Secure Discovery Service. In *Proc. ACM/IEEE MOBICOM*, pages 24–35, August 1999.

[20] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, December 1984.

[21] David Demirdjian, Konrad Tollmar, Kimberle Koile, Neal Checka, and Trevor Darrell. Activity maps for location-aware computing. In *IEEE Workshop on Applications of Computer Vision (WACV2002)*, Orlando, Florida, December 2002. IEEE.

[22] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16:97–166, 2001.

[23] Anind K. Dey, Masayasu Futakawa, Daniel Salber, and Gregory D. Abowd. The conference assistant: Combining context-awareness with wearable computing. In *3rd International Symposium on Wearable Computers (ISWC)*, pages 21–28, October 1999.

[24] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[25] Richard Fikes. Monitored execution of robot plans producted by STRIPS. In *Proceedings of the IFIP Congress*, volume 1, pages 189–194, Ljubljana, Yugoslavia, August 1971.

[26] Richard Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(1–3):251–288, 1972.

[27] R. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 49–54, Chicago, Illinois, June 1994.

[28] Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS 2001*, 2001.

[29] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, April-June 2002.

[30] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth International Conference on Artificial Intelligence Planning Systems*, pages 802–815, San Jose, California, July 1992.

[31] Robert Grimm. *System support for pervasive applications.* PhD thesis, University of Washington, December 2002.

[32] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, August 2001.

[33] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, pages 22–28, Callicoon, NY, June 2002. IEEE Computer Society Press.

[34] Eric Horvitz, Jack Breese, David Heckerman, David Hovel, and Koos Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265, Madison, WI, 1998.

[35] Sun's Java XML encoder (class named XMLEncoder). http://java.sun.com/j2se/1.4.1/docs/api/.

[36] JDOM Project home page. http://www.jdom.org.

[37] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–75, April-June 2002.

[38] Miryung Kim, Gary Look, and João Sousa. Planlet: Supporting plan-based user assistance. Technical Report IRS-TR-03-005, Intel Research Lab, Seattle, 2003.

[39] Ajay Kulkarni. A Reactive Behavioral System for the Intelligent Room. Masters of Engineering Thesis, MIT Artificial Intelligence Laboratory, 2002.

[40] Justin Lin. Personal location agents for communicating entities. Masters of Engineering Thesis, MIT, Cambridge, MA, 2002.

[41] Gary Look, Stephen Peters, and Howard Shrobe. Plan-driven ubiquitous computing. Ubicomp 2003. *In submission.*

[42] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999.

[43] Michael McGeachie and Jon Doyle. Efficient utility functions for ceteris paribus preferences. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 279–284, Edmonton, Alberta, Canada, July 2002. AAAI.

[44] Brad Myers, Robert Malkin, Michael Bett, Alex Waibel, Ben Bostwick, Robert C. Miller, Jie Yang, Matthias Denecke, Edgar Seemann, Jie Zhu, Choon Hong Peck, Dave Kong, Jeffrey Nichols, and Bill Scherlis. Flexi-modal and multi-machine user interfaces. In *IEEE Fourth International Conference on Multimodal Interfaces*, pages 377–382, Pittsburgh, PA, October 2002.

[45] Brad A. Myers. Using hand-held devices and PCs together. *Communications of the ACM*, 44(11):34–41, November 2001.

[46] H. Penny Nii. Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, 1986.

[47] H. Penny Nii. Blackboard systems, part two: Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, 7(3):82–106, 1986.

[48] Alice Oh, Harold Fox, Max Van Kleek, Aaron Adler, Krzysztof Gajos, Louis-Philippe Morency, and Trevor Darrell. Evaluating Look-to-Talk: A gaze-aware interface in a collaborative environment. In *Proceedings of CHI 2002*, Minneapolis, MN, April 2002.

[49] Charles L. Ortiz, Jr and Barbara J. Grosz. Interpreting information requests in context: A collaborative web interface for distance learning. In Michael Wooldridge and Katia Sycara, editors, *Autonomous Agents and Multi-Agent Systems*, volume 5:4. Kluwer, December 2002.

[50] Sharon Oviatt, Phil Cohen, Lizhong Wu, John Vergo, Lisbeth Duncan, Berhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and David Ferro. Designing the user interface for multimodal speech and pen-based gesture applications: State-of-the-art systems and future research directions. *Human Computer Interaction*, 15(4):263–322, 2000.

[51] Randy Pausch, Tommy Burnette, A.C. Capeheart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, and Jeff White. Alice: Rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications*, May 1995.

[52] Stephen Peters and Howie Shrobe. Using semantic networks for knowledge representation in an intelligent environment. In *PerCom '03: 1st Annual IEEE Inter-*

national Conference on Pervasive Computing and Communications, Ft. Worth, TX, USA, March 2003. IEEE.

[53] Martha Pollack and John F. Horty. There's more to life than making plans. *The AI Magazine*, 20(4):71–84, 1999.

[54] Martha Pollack and Colleen McCarthy. Towards focused plan monitoring: A technique and an application to mobile robots. In *IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, 1999.

[55] Martha E. Pollack, Colleen E. McCarthy, Ioannis Tsamardinos, Sailesh Ramakrishnan, Laura Brown, Steve Carrion, Dirk Colbry, Cheryl Orosz, and Bart Peintner. Autominder: A planning, monitoring, and reminding assistive agent. In *Proceedings of the Seventh International Conference on Intelligent Autonomous Systems*, March 2002.

[56] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of Ubicomp 2001*, Atlanta, Georgia, September 2001.

[57] Charles Rich, Neal Lesh, Jeff Rickel, and Andrew Garland. A plug-in architecture for generating collaborative agent responses. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Bologna, Italy, July 2002.

[58] Charles Rich, Candace L. Sidner, and Neal Lesh. Collagen: Applying Collaborative Discourse Theory to Human-Computer Interaction. *Artificial Intelligence Magazine*, 22(4):15–25, Winter 2001.

[59] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October-December 2002.

[60] Howard Shrobe. Computational vulnerability analysis for information survivability. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 919–926, Edmonton, Alberta, Canada, July 2002. AAAI.

[61] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[62] Max Van Kleek. Intelligent environments for informal public spaces: the Ki/o kiosk platform. Masters of Engineering Thesis, Massachusetts Institute of Technology, February 2003.

[63] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-based monitoring for continuous planning in dynamic environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 171–179, Pittsburgh, PA, June 1998.

[64] Vision Interface Project home page. http://www.ai.mit.edu/projects/vip/.

[65] Zhenyu Wang and David Garlan. Task-driven computing. Technical Report CMU-CS-00-154, Carnegie Mellon University, May 2000. http://reports-archive.adm.cs.cmu.edu/cs2000.html.

[66] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.

[67] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[68] Brian Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, August 1996.

[69] Brian Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, August 1997.