

# Autonomic Pervasive Computing based on Planning

Anand Ranganathan, Roy H. Campbell  
University of Illinois at Urbana-Champaign  
{ranganat,rhc}@uiuc.edu

## Abstract

*Pervasive Computing envisions a world with users interacting naturally with device-rich environments to perform various kinds of tasks. These environments must, thus, be self-managing and autonomic systems, receiving only high-level guidance from users. However, these environments are also highly dynamic – the context and resources available in these environments can change rapidly. They are also prone to failures – one or more entities can fail due to a variety of reasons. The dynamic and fault-prone nature of these environments poses major challenges to their autonomic operation. In this paper we present a new paradigm for the operation of pervasive computing environments that is based on goal specification and STRIPS-based planning. Users as well as application developers can describe tasks to be performed in terms of abstract goals and a planning framework decides how these goals are to be achieved. This paradigm helps improve the fault-tolerance, adaptability, ease of programming and usability of these environments. We have developed and used a prototype planning system within our pervasive computing system, Gaia<sup>1</sup>.*

## 1. Introduction

Pervasive computing advocates the construction of large distributed systems that feature a number of devices and services. These devices and services are meant to help users perform various tasks more easily and efficiently. Besides, these devices are supposed to disappear into the surroundings and not intrude on the user's consciousness. This requires pervasive computing environments to be self-managing and autonomic, requiring minimal user intervention.

At the same time, these environments are also highly dynamic and fault-prone. New kinds of entities (services, devices or applications) can enter these environments at any time. Existing entities may fail or leave the environment. The context of these environments (location of users, their activities, etc.)

can also change. Entities in the environment can fail due to a variety of reasons : hardware faults, OS errors, software bugs, network faults, etc. The dynamic and fault-prone nature of these environments makes it necessary to design new techniques to ensure the smooth operation of these environments.

In this paper, we present a new paradigm for the operation of pervasive computing environments that is based on goal specification and AI planning techniques. This paradigm allows users to specify their goals in an abstract manner and let the environment decide how best to achieve these goals. The paradigm makes use of a planning framework that plans how to achieve abstract user goals. It makes use of user's preferences, the current context and security policies to decide what would be the "best" final state of the environment, in which the user's goals are satisfied. It, then, makes use of AI planning to get a sequence of actions that transform the environment from its current state to the "best" final state. The framework, then, executes this sequence of actions to take it to the desired final state. While executing the actions, it monitors the effects of the actions through various feedback mechanisms. In case any of the actions fail, it handles the failure by retrying the same action or re-planning to come up with an alternate path to achieve the same goals.

The planning framework is based on a predicate model of pervasive computing, where the state of an environment is represented as a set of predicates. There is also a model of the actions that can be performed in the environment. An action is an invocation of a method on a service, device or application. A utility function is used to determine the "best" goal state. Apart from end-users, application-developers can also use the planning framework, through an API, to specify tasks that their application has to perform in terms of abstract goals.

The use of planning enables the environment to decide, dynamically, how best to achieve user goals. It relieves users of the burden of having to know exactly what can or cannot be done in an environment and how to perform various tasks. More importantly, since the environment, dynamically, plans the sequence of actions to achieve the user's goals, it can adapt more easily to changing contexts and availability of resources. It also allows the environment to handle faults that may occur

---

<sup>1</sup> This research is supported by a grant from the National Science Foundation, NSF CCR 0086094 ITR and NSF 99-72884 EQ

while achieving the user's goals gracefully and with minimal user intervention.

The planning framework has been implemented within Gaia[2], our pervasive computing system. Section 2 describes a motivating scenario. Section 3 describes the features of the framework and Section 4, the planning algorithm. Section 5 describes our environment model and Section 6 goes in to details of the planning algorithm. Section 7 describes the architecture of the planning system. We evaluate our approach in Section 8. Sections 9 and 10 have related work, future work and our conclusions.

## 2. Scenario

In order to motivate the planning paradigm, we describe a simple scenario. Assume that a user enters a smart room with the goal of starting a presentation. A major difficulty facing the user is choosing the best way of achieving this goal. Pervasive computing environments feature a large number of devices, applications and services. Our prototype smart room, for instance, allows presentations to be displayed on various plasma displays on different walls, a video wall, touch screens, handhelds, etc. The presentation can be controlled using voice commands (by saying "start", "next", etc.) or using a GUI (which has buttons for start, next, etc.) on his handheld or on various screens. Different applications (like Microsoft PowerPoint or Acrobat Reader – after converting the file to pdf format) can be used as well for displaying the slides.

Some ways of achieving the goal may be better than others because of context or user preferences. Some ways may even be prohibited because of access control restrictions. The best way of achieving the goal may also change with time because of availability of different devices or changing contexts. Also, depending on the current state of the environment, different actions may need to be taken to achieve the goal. Hence, it is not easy to statically specify how the goal is to be achieved. Besides, applications and devices may fail due to a number of reasons. We, thus, require techniques to simplify the environments for users to achieve their goals.

The Gaia planning framework helps solve some of the problems associated with the complexity, dynamism and fault-proneness of pervasive environments. The user indicates his goal ("display presentation") to the planning framework through a GUI. The framework then analyses the different choices available for achieving user goals and based on the user's preferences, the current context and access control policies, it chooses what it thinks is the best way of achieving the goal. It then plans a sequence of actions to achieve the goal and executes them. In this scenario, the user may have indicated a preference to using his

handheld device for controlling presentations. He may also prefer using Microsoft PowerPoint since he uses animations extensively. The current context of the environment is that there is to be a lecture in the room and there are a large number of attendees sitting in different parts of the room. Hence, the planning system may display the presentation using MS PowerPoint on all the plasma screens on the different walls and allow the user to control it using his handheld.

Let us now assume that some of the actions taken by the planning framework fail. For example, the controller on the user's handheld could not be started because of a transient failure of the wireless network. The framework detects the failure of this action by the fact that the handheld device is not reachable or by detecting that the controller, which it started, was not detected automatically by the Gaia Presence Service. The planning framework first tries to re-execute the action by contacting the handheld device again. If this action fails again, it re-plans to discover an alternate goal state that does not involve the handheld. The next best goal state may involve starting the controller in a touch screen close to the user. The planning framework then executes actions to achieve this new goal. If the new goal state is again unreachable, it tries the next best goal state and so on. All the while, it keeps sending feedback to the user giving details of the plan and the failures. If all goal states lead to failures, it informs the user that the goal is not currently reachable.

## 3. Features of the Planning Framework

Our planning framework offers a number of advantages in the development, deployment and use of autonomic pervasive environments.

*1. Improves usability.* The framework allows users to specify their goals in a simple manner and it automatically carries out actions to achieve the goal, or guides the user through a sequence of steps to achieve the goal.

*2. Improves ease of programming.* The planning framework also allows developers to describe the tasks to be performed by their applications in terms of abstract goals. This means that they do not have to worry about knowing how exactly to perform certain kinds of tasks in a distributed system, what kinds of services and applications are present in the system and how to interact with them. They can leave the intricate details of performing tasks as well as handling failures to the planning framework.

*3. Improves Fault-tolerance.* Objects in distributed systems can fail due to a variety of reasons – OS faults, software bugs, hardware errors, human error, etc. Hence, one or more actions required to perform a task may not succeed. If an action fails, the framework

detects it and recovers by retrying actions or by replanning and taking another path to achieve the same goal. The replanning approach to failure recovery works because pervasive environments are device-rich. Thus, even if one or more devices or applications fail, there are, normally, other ways of achieving the same goal.

4. *Customized to user preferences and context.* There may be many alternative states or configurations of the environment that satisfy the goals. For example, in the presentation scenario, there were many possible ways of displaying the presentation. The planning framework chooses a goal state that maximizes the utility of the user. This utility function is based on the user's preferences and the current context.

5. *Adaptable to different configurations* The planning framework allows tasks to be performed even in different configurations of the environment. The framework has a discovery component where it finds out what resources are currently available in the system and then tries to achieve the goal using these resources.

6. *Security.* The framework ensures that the goal states or actions do not violate access control policies.

7. *Portability to other environments.* Since applications and services developed using the planning framework perform tasks by expressing them in the form of abstract goals, they are not dependent on specific features of any environment. This allows them to be ported to other environments easily, and also increases reliability. The planning framework takes care of actually achieving the abstract goals of the application or service in the system it is deployed in.

## 4. Planning Algorithm

Briefly, the planning algorithm takes in an abstract goal specification, generates a template goal state and then converts that into a concrete goal state. It then plans actions to get to the goal state and executes them while handling failures. It interleaves the steps of discovery, goal-state generation, planning, execution and monitoring. In further detail, the steps are:

1. Get the goal(s) from the user or application
2. Generate final states where the goal(s) is satisfied using templates and choose the best final state using the utility function.
3. Get current state and load into STRIPS planner
4. Use STRIPS-based planner to get path from current state to chosen goal state
5. If no path is found, try again with the next best goal state generated in step 2. Repeat until we find a path to some final state where the goal is satisfied. If no path is found to any final state, exit and inform the user or application that the goal is not satisfiable.
6. If a path is found, then for each action in the path do the following:

- a. Execute action by calling the method on the service, device or application with the appropriate parameters
- b. Get feedback about the success of the action.
- c. If action succeeded, continue with next one.
- d. If action failed, do one of the following depending on type of action:
  - i. If the action is annotated as a retryable action, try executing the action again.
  - ii. Else, abort the execution of actions and go back to Step 2 to replan and to get a new goal state and path.

## 5. Model of the environment

The planning framework is based on a predicate model of the environment where states of the environment and of different entities are represented as sets of predicates. The state of the environment is the aggregate of the states of all entities (i.e. services, devices and applications) in the environment along with the context of the environment. Context is any information that is relevant to the interaction between the user and the environment and to the achievement of the user's goals. It includes information such as location of people, their activity, etc.

### 5.1. States of devices, services and applications

The state of each device and service in the system is represented as one or more predicates. For example; the state of a light bulb may be represented as `lightBulb(bulbID, room3232, off)`. The state of a Location Service may be `locationService(serviceID, cs_building, running)`. Each device and service supports a query interface that allows other objects to query it about its current state

Applications in Gaia consist of 4 main components based on the Model-View-Controller pattern[5]. These are a model (which manages application state), a presentation (that provides a view of the state), a controller (that allows users to control the application) and a coordinator (that has information about the structure of the application and its components). The state of an application is essentially the aggregate of the states of its components. For example, the state of a PowerPoint application may be:

```
coordinator( appID, pptApp, machine1,
initialized, [pres1, pres2], [cont1])
pptModel( appID, machine1, started,
"gaia.ppt", 1)
presentation( appID, pres1,
mspowerpoint, machine5)
presentation( appID, pres2,
mspowerpoint, machine6)
controller( appID, cont1,
handheldController, ipaq1)
```

These predicates state that the coordinator is initialized in machine1 and the application has two presentations pres1 and pres2 and one controller cont1. The model of the application is in machine1 showing the file “gaia.ppt” and is currently on slide number 1. Presentation pres1 is of type Microsoft PowerPoint and is running in machine5. Presentation pres2 is also of type Microsoft PowerPoint and is running in machine6. Controller cont1 is of type handheldController and is running on the handheld device ipaq1.

Context is also represented as predicates. Example context predicates are:

```
location ( chris , inside , room 3231)
temperature ( room 3231 , "=", 98 F)
```

The set of predicates that constitutes the state of a device, service or application as well as the structures of different predicates are specified in an ontology. The ontologies are written in DAML+OIL[4], one of the standard languages of the Semantic Web. The ontology is used to type-check predicates. It also makes it easier to use different predicates in developing rules, preferences or access control policies since we know what the structure of the predicate is and what kinds of values different arguments can take.

## 5.2. Model of Actions

Actions transform the state of the environment. We describe actions in PDDL[13]. Each entity in Gaia is associated with a PDDL file that describes the actions it can perform. The PDDL file is like an enhanced IDL file – along with the methods that can be invoked on the entity, it also describes the preconditions and effects of actions (method invocations). Since many standard planners understand PDDL, we can experiment with different planners (which have different powers, expressiveness and performance levels) to see which is appropriate for a pervasive computing domain.

For example, the following PDDL piece describes the setFile action for the PowerPoint application:

```
(:action setFile_ppt
  :parameters (?id - string ?filename
  - pptfile)
  :vars (?mac - machine ?slidenum
  - number)
  :precondition
    (and (pptModel ?id ?mac stated
    ?filename1 ?slidenum)
    (not (= ?filename ?filename1)))
  :effect
    (pptModel ?id ?mac started
    ?filename 1)
  :check
    (= (getFile_ppt ?id) ?filename)
    (failure non-retryable))
```

The above segment of PDDL states that the setFile\_ppt action has two parameters : the id of the application and the name of the file. It states that the precondition is that the model of the powerpoint application should be started on some machine and should be displaying a file that is different from the input file in the parameter. The effect of the action is that model now displays the input file and is on slide number 1.

Actions have additional information to help decide whether they have failed and what to do in case of failure. This information is specified through extensions to PDDL. Since an action is a method invocation, we use the return value of the invocation to check for failure. Another method to check for failure involves querying a certain service or application. For example, if the action is to start a new component on some machine, we can check if the component really started by querying the Space Repository (which maintains a list of all running entities in Gaia). In the above example, the check is performed by querying the application for the name of the current file and checking to see if it is the correct file.

If an action is marked as “retryable” in the PDDL file, the planning system tries to execute the action again if it failed the first time. We found that this approach is especially useful when controlling hardware (like lights and cameras) which may not respond the first time or when communicating with mobile devices (via 802.11 or Bluetooth, which are sometimes unreliable). Most other actions are marked as “non-retryable”, which means that it most probably will not help to retry the action because failure of the action implies failure of the application or service. In such cases, the planning system replans and finds an alternative path.

## 5.3. Utility Function

Each state of the environment is associated with a utility to a certain user (or application). This allows the planning framework to compare different goal states and choose the one with the maximum utility. We define the utility of the entire environment as a linear combination of the individual utilities of the states of all the services, applications and devices running in it.

For example, if the goal of the user or an application is to present a ppt file, there are various possible goal states (corresponding to different configurations of the application – i.e. different positions of controllers and presentations). One of the predicates in the goal state is the controller predicate. Possible utilities of different configurations of the controller may be

```
controller (appID, cont1, handheldControl
ler, ipaq1):
  6 if activity(room, meeting)
  4 if activity(room, presentation)
```

```

controller(appID,cont1,touchScreenCont
roller,machine4): 3
controller(appID,cont1,touchScreenCont
roller,machine7): 2

```

This states that the utility of a controller for the application depends on the type of controller (handheldController or touchScreenController), the device on which it is instantiated (ipaq1, machine4 or machine7) and the current activity in the room. The utilities in our current model vary from -10 to +10.

The utility of the goal state is a linear combination of the utilities of all the services and applications relevant

to the goal. In the PowerPoint example, the utility of the goal state is a linear combination of the utilities of different predicates involved – the controllers, presentations, pptModel and coordinator.

Each user has preference files that specify his utility for various predicates in different contexts. In the absence of a specific preference regarding a certain predicate, we assume that its utility is 0 (which is in between -10 and +10). Application developers can also write rule files that indicate the utility of different states for their applications.

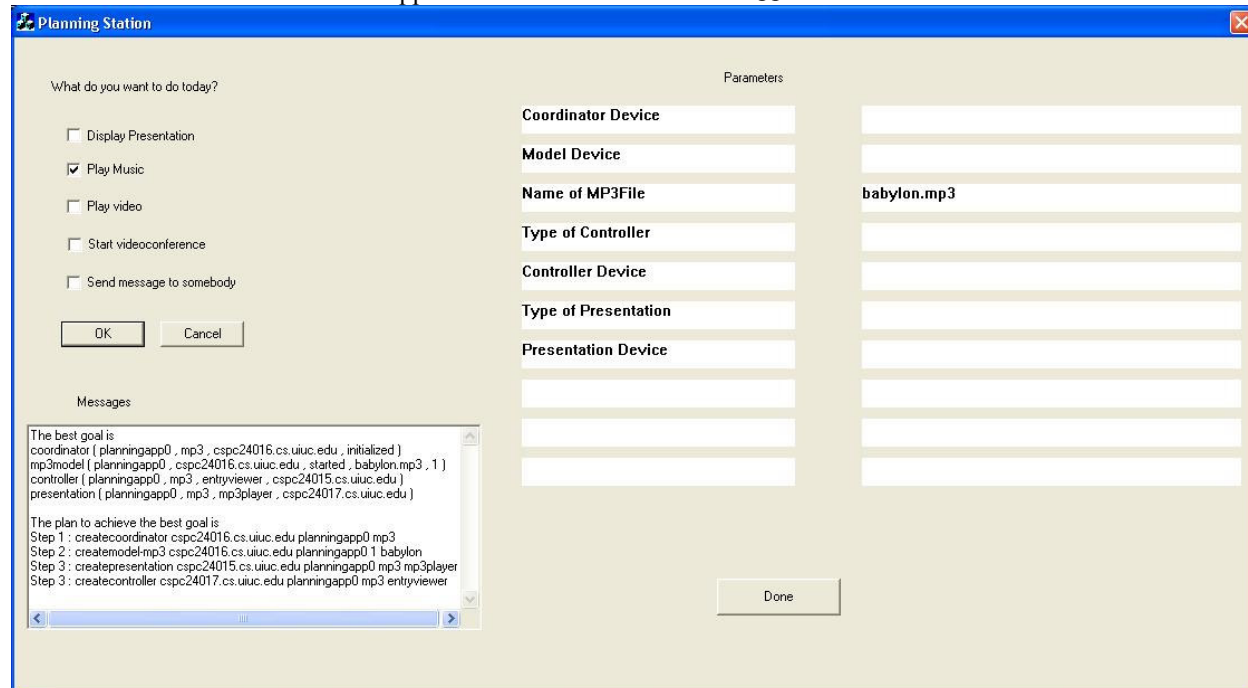


Figure 1. Screenshot of user interface

## 6. Steps of the Planning Process

The following subsections give details of the various steps in the planning process.

### 6.1. Getting the goal description

A user specifies his goal(s) through a GUI, which then calls the planning framework. Applications can also call the planning framework specifying goals in the form of one or more predicates.

A screenshot of the GUI is shown in Fig 1. The user can choose a goal (such as “play music”). He is then presented with the parameters for the goal. He can fill in some of the parameters (like the song to be played). He can also choose to leave other parameters (like the machines on which various components are to be started) blank and let the planning framework decide the best values for these parameters. The GUI then gives him feedback on the chosen best goal state (with the

goal parameters that the planning framework has filled in) and the execution of the plan.

### 6.2. Choosing the best goal state

Each goal is associated with template final states where the goal is satisfied. Rules for getting these template states are written in Prolog. The template states are represented as predicates having some variable arguments. Putting different values for these arguments gives us different final states where the goal is satisfied.

For example, if the goal is to display a powerpoint file, the template final state contains the coordinator, pptModel, controller and presentation predicates (and maybe others depending on what the goal is exactly). Variables in these predicates include the machines on which these components can be run, the types of controllers and presentations to use, the name of the file to display, etc. The end-user or the application can give more information as well to reduce the number of

variables. For example, he can specify the name of the file and thus remove that variable from the pptModel predicate. An example of such a rule is:

```
goal(display_ppt, File) :- coordinator(
  appID, pptApp, M1, initialized),
pptModel( appID, M2, started, File, 1),
presentation(appID, pres1,
  PresentationType,M3) ,
controller( appID, cont1,
  ControllerType,M4)
```

This rule, written in Prolog, states that goal of displaying a powerpoint file called File can be achieved if there is a coordinator of a ppt application initialized in some machine M1 and a pptModel started in machine M2 showing slide number 1 of File. A presentation and a controller should also be present. Arguments that begin with capital letters are variables that have to be instantiated by the goal state generator.

The planning framework first finds possible values that the variables can take. Variables in predicates have types; it makes use of this type information to get possible values. For example, if the variable is of type “machine”, it discovers all machines running in the system currently from a registry. If the variable is of type “ControllerType”, it queries the ontologies to get possible values that this variable can take.

The framework then chooses values of the variables that maximize the utility. Since we use a utility function that is a linear combination of the utilities of different predicates, the planning framework chooses variable values that maximize the utility of each predicate individually. The main assumption here is that each predicate is independent and the state with the maximum utility contains predicates with maximum utilities. The independence of predicates also implies that the same variable does not appear in more than one predicate. If there are many values that give the same utility of a predicate, one of them is chosen randomly.

The planning framework also checks the goal states and actions to ensure that they satisfy RBAC-based access control restrictions. Access control policies are specified as roles being allowed (or not allowed) to perform certain actions on a certain service or application in a certain context. For example, a policy could specify that a guest cannot perform the “setFile\_ppt” action on a PowerPoint application if the current activity in the room is “Presentation”. If the best goal state does not satisfy an access control policy, it considers the next best state and so on.

The stage of generating goal states is distinct from the planning phase because most standard planners do not have a notion of utility – they just discover a path to any goal state. Hence, we need to discover the best goal state beforehand and then use STRIPS-planning to get the path to it.

### 6.3. Getting the current state

The framework discovers all services, applications and devices running in the environment from a registry service and then queries all of them to get their current state. It also gets the current context of the environment from various sensors and other services.

### 6.4. Using STRIPS-planning to get path

The planning framework then invokes a standard planner to get a sequence of actions to transform the state of the system from the current state to the chosen final state. Our current implementation uses the Blackbox planner[1] that combines a graph-based planning approach with a SAT-based approach. The planner is initially loaded with all possible actions that can be performed in the environment from the PDDL files of different entities.

An example of a plan that is generated for achieving the goal of displaying a specific powerpoint file is shown below (in Lisp format):

```
1 (createcoordinator cspc24016 abc ppt)
2 (createmodel-ppt cspc24016 abc 1
  planning.ppt)
3 (createcontroller ipaq1 abc
  handheldController)
3 (createpresentation cspc24015 abc
  mspowerpoint)
3 (createpresentation cspc24017 abc
  mspowerpoint)
```

This plan indicates that the steps for satisfying the goal are to first create a coordinator in the machine cspc24016 with id “abc”; then, to create a model for the PowerPoint application in cspc24016 initialized with the file “planning.ppt”; then, to create a controller for it in the handheld ipaq1 and two presentations in the machines cspc24015 and cspc24017. The three actions with the same serial number 3 can be done in parallel.

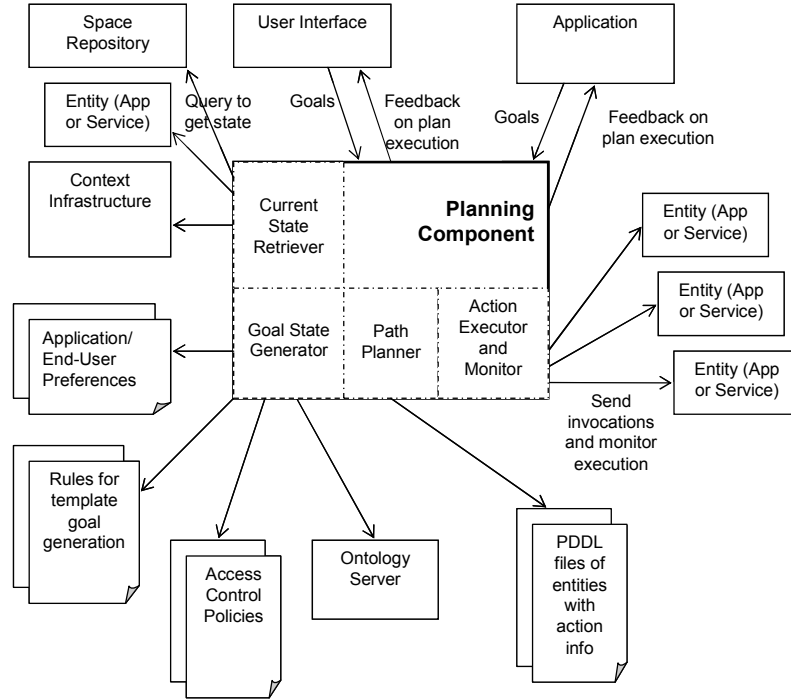
The steps are executed by invoking methods on various objects with the parameters specified in the step. For example, the first step involves invoking the createcoordinator method on the object representing the computer cspc24016 with the parameters “abc” and “ppt”. The types of the parameters are obtained from the ontologies (which contain all information about the structure of predicates).

### 6.5. Handling failures

A failure of an action is detected if the entity on which the method is invoked is not reachable or does not respond, or from the return value of the invocation (a return value less than 0 indicates failure). A failure can also be inferred by querying another service (as specified in the PDDL file). If a failure is detected and the PDDL file indicates that the action is retryable, the planning framework invokes the same action once

again, else it replans to get to the next best goal state and the path to it. Replanning, although a very simple solution, may be inefficient, since it involves completely discarding the old plan and starting all over again. However, replanning is still acceptable, since

most plans in our prototype pervasive environment are short in length (less than 15 steps and a few thousand states). So, the overhead in replanning is not significant. In the future, though, larger environments and plan lengths may force us to explore other methods.



**Figure 2. Architecture of the Planning Framework**

## 7. Structure of the Planning Framework

The key element of the architecture (shown in Fig 2) is the Planning Component. An application calls the Planning Component with its goals. A user communicates his goals to the Planning Component through a User Interface. The Planning Component has 4 key modules that perform different aspects of the planning process. The modular architecture helps separating different concerns and also allows us to use alternate algorithms or mechanisms for performing various parts of the process. The 4 modules are:

1. *Current State Retriever*, that gets the current state by querying the Space Repository (a registry in Gaia) and then queries the various entities (applications, services or devices) to get their current state. It gets the current context from the context infrastructure.

2. *Goal State Generator*, that generates possible goal states and chooses the best one, using preference information, access control policies, ontologies and the current state and context of the environment.

3. *Path Planner*, that finds the actual sequence of actions to bring the environment to the chosen goal

state, using action information from PDDL files and a STRIPS planner (Blackbox).

4. *Action Executor and Monitor*, that executes actions by calling methods on various objects (applications and services) and monitors them for failures. It makes invocations using CORBA's Dynamic Invocation Interface (which allows calling methods on remote objects dynamically without requiring pre-compiled stubs).

The Context Infrastructure[11] consists of distributed sensors and other components that deduce the current context in the environment. The Ontology Server[12] manages all the ontologies in Gaia and provides descriptions of classes of entities in the Space, meta-information about context, and other information in the ontologies. The Space Repository maintains a list of entities in the environment and their properties. New entities are discovered by a heartbeat mechanism[2].

## 8. Implementation and Evaluation

The Planning Component has been implemented as a library. It exports methods for specifying goals,

retrieving the current status of the plan execution, specifying preference files and so on. It can be run as a standalone object in Gaia or linked with other objects. Other objects in Gaia (like the User Interface GUI) invoke the Planning Component through local or remote method invocation. Remote methods are invoked using CORBA. We interact with various Windows applications (like PowerPoint and WinAmp) using their COM interface.

The planner has helped simplify various tasks for users as well as application developers. For instance, the bootstrapping of Gaia is a complex process involving starting up various services with interdependencies on different machines and there is always the possibility of failure as well. The framework has helped simplify this process by taking care of dependencies while starting services and handling failures as well. The dependencies are expressed in terms of preconditions in PDDL files. It has also helped reduce the effort required by users to perform common tasks in our smart room like controlling various applications, and collaborating with others.

In terms of performance, the overhead imposed by the planning framework is about 30% over a static script. For example, the planner took 5.65 seconds (for a plan of length 8) to bring up a presentation in a certain configuration, while a static script took 3.95 seconds. If an action fails and the planner has to abort and replan, the time taken to achieve the goal grows with the number of failures. The plan of length 8 took 5.65 seconds with no failures, 9.8 seconds with 1 failure and 13.6 seconds with two failures.

## 9. Related Work

There has not been much work in the use of planning for pervasive autonomic computing. MIT's Oxygen Project[6] automatically satisfies abstract user goals by assembling, on-the-fly, an implementation that utilizes the resources currently available to the user. However, their system has no mechanism for handling failures. There has been work done on the composition of web services using planning[7][9][10]. However, all these systems do not have mechanisms for monitoring the execution of plans and replanning in case of failures.

The iROS[8] system is based on an Event Heap and uses soft-state maintenance and fast restart to recover from failures. It, however, does not allow users to perform tasks in terms of abstract goals or discover alternate configurations for achieving goals in case of failures. There has been some work done in the area of replanning and fault-tolerant planning. The Lifelong Planning A\*[3] algorithm efficiently replans shortest paths upon changes in edge weights by reusing information from previous episodes.

## 10. Conclusions and Future Work

In conclusion, the paper makes use of AI planning to enable autonomic pervasive computing. The system helps user perform tasks with minimal intervention. It also simplifies the task of application developers in handling failures and performing various kinds of tasks.

In the future, we plan to investigate the issue of conflicting goals issued by multiple objects or end-users. The current framework can handle multiple users (and Planning Components), though, it does not attempt to find and handle possibly conflicting goals. We also need to carry out further user studies to validate the use of the approach as well as test the scalability of the system in larger environments. In the current design, failure of the planning framework cannot be handled. We plan to tackle this through redundancy. We also plan to use machine learning for automatically getting user preferences and their utility functions.

## 11. References

- [1] Kautz, H. and Bart Selman. "Unifying SAT-based and Graph-based Planning". Proc. IJCAI-99, Stockholm.
- [2] Román, M., et al, "Gaia: A Middleware Infrastructure to Enable Active Spaces". In IEEE Pervasive Computing, pp. 74-83, Oct-Dec 2002..
- [3] Koenig, S. et al. "Heuristic Search Based Replanning" In International Conf. on AI Planning and Scheduling, 2002
- [4] Harmelon, F., et al "Reference Description of the DAML+OIL ontology markup language", <http://www.daml.org/2001/03/reference.html>
- [5] Krasner G E and Pope S T (1988) "A description of the model-view-controller user interface paradigm in the smalltalk-80 system". Journal of Object Oriented Programming 1:26-49
- [6] Saif U., et al "A Case for Goal-oriented Programming Semantics". In System Support for Ubiquitous Computing Workshop at Ubicomp, Seattle, WA, Oct 12, 2003
- [7] McIlraith, S. and Son, T.. Adapting Golog for Composition of Semantic Web Services. KR2002, Toulouse, France, 2002.
- [8] Ponnekanti, S.R. et al. "Portability, Extensibility and Robustness in iROS", PerCom 2003.
- [9] Sheshagiri, M. et al. A Planner for Composing Services Described in DAML-S. In Workshop on Planning for Web Services, International Conference on Automated Planning and Scheduling, Trento, July 2003
- [10] Hendler, J. et al. Automating DAML-S Web Services Composition using SHOP2. In ISWC2003 Florida, Oct 2003
- [11] Ranganathan, A., Campbell, .R.H.. "A Middleware for Context-Aware Agents in Ubiquitous Computing Environments" In Middleware 2003, Rio de Janeiro, Brazil,
- [12] Ranganthan, A. et al. "Ontologies in a Pervasive Computing Environment". In Workshop on Ontologies and Distributed Systems (IJCAI 2003), Acapulco, Aug 9 2003
- [13] McDermott,D., and the AIPS-98 Planning Competition Committee. "PDDL - The Planning Domain Definition Language", Draft 1.6, June 1998