OS Agents: Using AI Techniques in the Operating System Environment

Oren Etzioni, Henry M. Levy, Richard B. Segal, and
Chandramohan A. Thekkath

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 93-04-04
August 3, 1994

OS Agents: Using AI Techniques in the Operating System Environment

Oren Etzioni, Henry M. Levy, Richard B. Segal, and
Chandramohan A. Thekkath
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

August 3, 1994

Abstract

While recent decades have brought substantial change to the *form* of the operating system interface, the *power* of operating system commands has remained nearly constant. Conventional commands, whether visual or textual, specify one particular action to perform. To carry out a complex task, such as reducing disk utilization, the user is forced to explicitly specify each of the necessary steps. Traditional command-language extension mechanisms, such as shell scripts in Unix, enable the user to aggregate and compose various commands, but force him or her to write and debug programs — a formidable challenge for naive users.

This paper presents a *goal-oriented* approach to the operating system command interface, realized through an implementation we call OS agents. Using OS agents, the user simply specifies a goal to accomplish, and the OS agent decides how to accomplish that goal using its knowledge base of the system state and its commands. The OS agent dynamically synthesizes the appropriate command sequence, issues the required commands and system calls, handles errors, and retries commands if necessary. With OS agents, we have applied AI planning and learning techniques to the operating system environment to increase the power of the user's commands.

We have implemented OS agents within a distributed Unix environment. Our experience indicates that it is practical to incorporate novel ideas of automatic planning and learning into contemporary operating systems with a modest amount of work and little performance penalty. In this paper we present OS agents and their operation, and describe the general-purpose mechanism we have provided to flexibly and efficiently support the needs of OS agents within the Unix operating system.

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, CCR-9200832, and IRI-9211045), Office of Naval Research Grant 92-J-1946, the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. Oren Etzioni is supported in part by an NSF National Young Investigator Award. C. Thekkath was supported in part by a fellowship from Intel Corporation. H. Levy was supported in part by a Fulbright research award and by INRIA. Richard Segal is supported by a GTE fellowship.

1 Introduction

While previous decades have seen substantial change in the form of the operating system user interface, little has changed in the nature of what users can express. That is, while modern (e.g., menu-based) interfaces make it easier for naive users to issue basic requests, they do not reduce the difficulty of specifying more sophisticated queries or operations. Sophisticated requests will always require users to combine or customize system-provided primitives, often in an ad hoc way, because command language and interface designers cannot anticipate all possible user needs. The result is that many conceptually simple requests may be out of the reach of the naive user.

This paper presents a new *goal-oriented* approach to the operating system command interface that attacks this problem. This approach enables the user to request tasks that while simple to specify, may be complex to carry out with conventional commands. For instance, here are some examples of user requests for which no primitives are available in typical modern operating systems:

- **Event:** Mail me any posts containing the string "bicycle" as soon as they appear on the market bulletin board this week.
- **Constraint:** Keep my schedule consistent with the departmental visitor schedule.
- **Script:** Every morning retrieve my mail from the machine called June, start up my news reading program, and display the weather forecast.
- **Set specification:** Compress all Postscript files in this directory that have not been modified in more than six months.

While such idiosyncratic commands cannot be primitive in any operating system interface, such requests should be possible for even the *naive* user. A number of approaches to this problem have been adopted in previous work on operating system interfaces. The most common is a command language extension mechanism, such as Unix shell scripts. However, this approach forces naive users to write and debug programs—a formidable challenge. As a specific example, the standard command interface to Unix-like operating systems is a form of programming language, but an impoverished language with little or no program development environment. As another approach, some interfaces contain preset configuration menus, enabling the user to state her preferences. Such menus are necessarily restricted to situations that the application designer is able to anticipate. Furthermore, configuration menus do not easily facilitate the composition or combination of multiple commands. The third more recent approach is "programming by demonstration," where the user demonstrates what she is trying to achieve using an example, and the interface attempts to infer the general pattern that the user desires. User demonstrations are inherently ambiguous, however, thus the interface confronts a difficult inference problem in generalizing from the demonstration to the user's general goal.

In this paper we present a new goal-oriented approach to the operating system command interface. A goal-oriented interface enables a user to specify *what* to accomplish, leaving the decision of *how* to accomplish it to the system. While this idea is widely accepted in the functional and logic programming communities, current operating system interfaces do not support it. Although an interface may have a command (or shell program) ready to handle a *particular* idiosyncratic request, it cannot supply commands to support every conceivable user goal or combination of goals. Currently, the user is forced to write a script

or program by composing commands. In contrast, the key feature of our approach is that it obviates *any* programming by the user.

This paper describes the design and implementation of *OS agents* that satisfy high-level user requests. We use the agent terminology in loose analogy to a travel agent or insurance agent who, given a goal, searches for a means to accomplish it. The OS agent possesses a database that describes available system primitives (e.g., system calls, system programs, and shell scripts); this database defines the required preconditions and inputs, and the produced postconditions and outputs, for inovcation of those system primitives. Given a user-specified goal that can be accomplished by requests on those primitives, the OS agent constructs and issues a sequence of invocations to accomplish that goal, making decisions along the way based on intermediate results and conditions if needed. The agent is thus free to choose the most effective means of accomplishing a particular task, relying on commands or information that the user may not even be aware of (e.g., printer-3 is down today).

To achieve the level of functionality that we require, an OS agent must be able to perform the following:

- *Dynamically synthesize a sequence of OS commands, system calls, or shell programs to accomplish a wide range of user goals.*
  We insist that the sequence be synthesized dynamically, instead of statically defined, because of the large (potentially infinite) set of tasks the user can express. Furthermore, different circumstances can dictate differing responses to the very same goal. The OS agent should dynamically choose the appropriate actions based on the current system configuration. Thus, hard-coding the appropriate responses ahead of time is impractical. We discuss this in greater detail in Section 2.

- *Execute OS commands, recover from any unexpected failures, and automatically attempt alternative methods for completing the task if necessary.* For example, if the network crashes during a lengthy file transfer, the OS agent should be able to restart the transfer, when appropriate, and even attempt to retrieve the desired file from a different source, if necessary.

- *Adapt to differing conditions, resources, and user tastes.*
  The OS agent has to be customized to its individual users and to conditions prevailing in its operating environment. To reduce disk utilization, for example, one user may prefer to compress ancient files, another may wish to transfer files to a different disk, and a third may want to delete large postscript files.

While designing and implementing an agent to do this may sound difficult, in fact, the base technology already exists within two major areas of Artificial Intelligence (AI) known as *planning* and *machine learning*. In recent years, concise algorithmic descriptions and public-domain implementations have become available, attesting to the maturity of the techniques. Furthermore, the agent is poised to leverage new advances: as new learning and planning algorithms become available, we can plug them into the agent improving its competence. We have designed and implemented a prototype OS agent, called the Internet Softbot, which has been in use at the University of Washington for over a year.

We do not claim that our approach is necessarily better, but merely that it is worth exploring and understanding the trade-offs between our approach and existing approaches. Notice that while our approach has some resemblance to "automatic programming," we restrict the types of goals that a user is allowed to specify. Figure 1 contrasts our approach with the traditional approach of using customized programs to interface to the operating system.

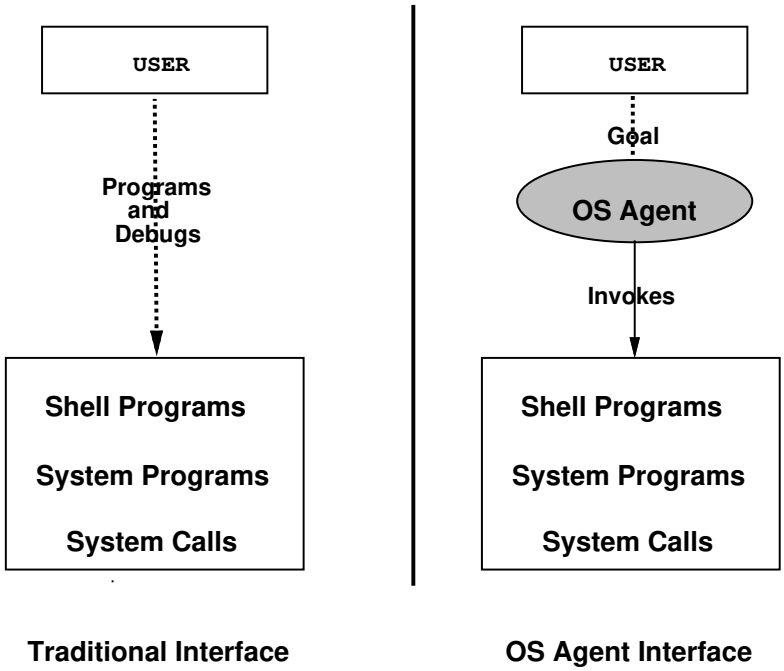

Figure 1: OS Interfaces: In the traditional interface, the user writes and debugs a customized script as a composition of other shell programs, system programs, and system calls. In the OS Agent interface, the user specifies a goal that is then automatically satisfied by the OS Agent.

It is instructive to compare OS agents with more standard OS interfaces. Although a competent system programmer could write a system program or shell script to satisfy any *individual* user request that can be handled by OS agents, OS agents have a number of advantages over the traditional interface:

- The programmer would need to write a program for *every* composite goal that a user might wish to express, whereas the OS agent accepts arbitrary logical combinations of primitive subgoals and automatically decomposes them into their constituents.
- The programmer could attempt to develop a set of primitive programs and expect the user to compose them, relying on a mechanism such as Unix pipes. However, this approach would yield a facility far less powerful than OS agents, which would still burden the user with programming the OS command interface.
- Shell programs are committed to a rigid control flow, determined a priori by the programmer. Yet, writing programs that anticipate and adapt to all possible changes in system environment, error conditions, and so on, is extremely difficult. In contrast, once a user specifies a goal, the OS agent *dynamically* generates a sequence of commands to satisfy it, fluidly backtracking from one option to another based on information collected at run time. The OS agent's response to a particular goal can change, depending on transient system conditions.
- We cannot expect the system programmer to modify his shell programs to suit each individual user. Thus, the burden of customizing a potentially large collection of programs would fall on the individual

user, as is the case with existing systems. In contrast, the OS agent's control heuristics provide a natural and flexible means of customizing the agent's backtracking search to different users. For example, the agent could learn that `whois` *never* finds a particular user's acquaintances, and save time by abstaining from that command. Simple, but effective, heuristics of this sort can be generated automatically using machine learning techniques [5, 8].

- To enable shell programs to utilize a new command, or a new application, the programmer would have to change *all* programs that might potentially use the command. In contrast, models of new command or tools can be easily added to the agent's database as they become available. More important, given logical command models describing a new application, the agent can *immediately* begin to access the application and incorporate this capability into *all* of its plans, for all user goals.

To illustrate the extensibility of our approach, consider the task of finding a user's e-mail address given his name (a common task for our agent). To do this, the agent employs models of the obvious facilities, such as `finger` and `whois`, as well as more esoteric heuristics, such as searching for the user's name on bboards, in bibliography files, and in old mail messages. Recently, we discovered the `netfind` facility distributed by the University of Colorado [11]. We added three command models to our agent's repertoire, and it is now able to utilize `netfind` to locate users. The addition of this capability required only addition of the command models for `netfind` to the agent's database, no programming; yet, `netfind` can now be used as part of the plan to satisfy any goal.

In short, although one can imagine building a shell script or system program that exhibits *some* of the above features for specific, individual tasks, our agent provides them as general facilities available to support arbitrary tasks in a uniform and extensible manner. To match the power of OS agents with customized shell scripts, e.g., as in Unix, a user or system programmer would need to create programs to accomplish every conceivable user goal or combination of goals. Furthermore, should a new system facility become available, each shell script would need to be modified to use it.

In this paper, we are demonstrating the marriage of what may seem like strange bedfellows—AI and operating systems. The organization of the paper follows the two different viewpoints from which this work can be seen. In Section 2 we provide a high-level description of the AI aspects of OS agents; this includes goal specification, planning of actions, and representation of knowledge. We cannot possibly hope to supply a comprehensive account of the relevant AI technology within this paper; instead, we present simple examples that give an intuitive feel for the approach and provide pointers to the AI literature. Section 2.4 shows the planning process for a simple goal and how it is accomplished.

In Section 3 we discuss our support for OS agents within Unix. A facility such as OS agents requires specialized support from the operating system if it is to be effective and efficient. Specifically, OS agents rely on event notification and general informational services. Since it is impossible to anticipate all types of user requests, the underlying support system must be extensible, so that new types of information or new events can be easily added. Thus, while some information and event services exist already within all operating systems, our goal was to prototype a general, extensible service on which OS agents could call for their needs. Finally, Section 4 presents some performance measurements of the system, and Section 5 concludes.

2 An OS Agent for UNIX

This section describes the overall architecture and the components of the OS agent. At a high level, the agent's architecture is decomposed into the following modules:

- The **Goal Manager** receives task specifications from the user, and periodically invokes the planner with goals to achieve.
- The **Planner** synthesizes sequences of OS commands to satisfy requests received from the user. Commands to be executed are sent to the OS Manager.
- The **Model Manager** serves as the central repository for the OS agent's knowledge. The model manager stores logical propositions describing OS commands, as well as the system's state (such as what files are in the directory `bin`). Our model manager employs standard knowledge-representation techniques and will not be discussed further in this paper.
- The **OS Manager** handles all interaction between the OS agent and its external environment, including inter-node communication, command execution, status checking, and so on. The design and implementation of the OS Manager is described in Section 3.

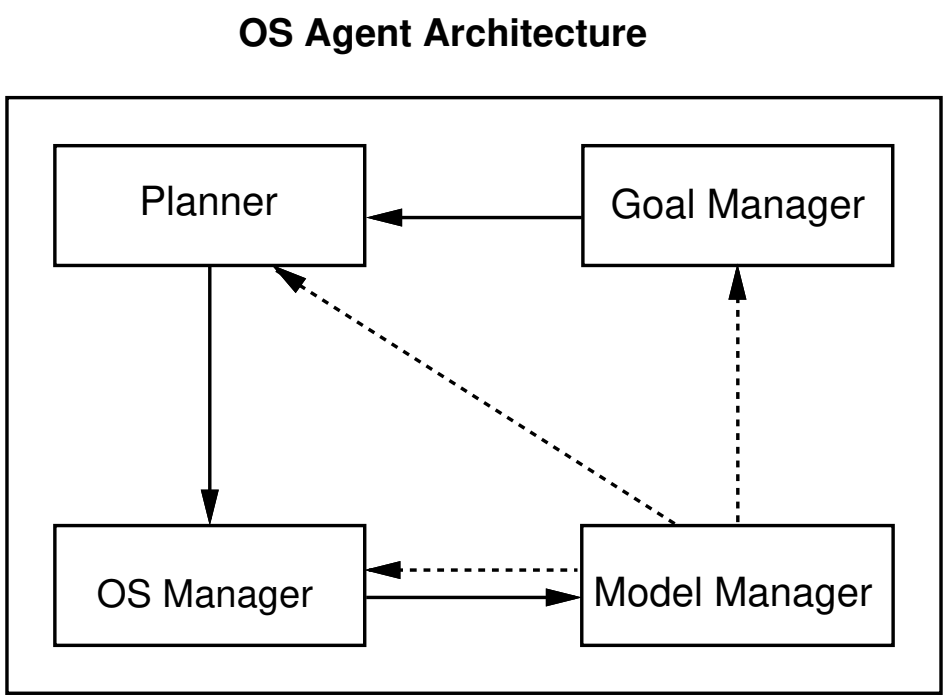

Figure 2: Architecture of the OS agent

Figure 2 shows the architecture of our OS Agent. The solid arrows represent control flow, and the dashed arrows indicate data flow.

In the first subsection below, we present examples of the goal language to indicate how a user could specify goals. Second, we describe the OS agent's goal manager, focusing on how the goal manager decomposes user goals into a series of calls to the planner. Third, we present the planner's algorithms and explain the formal guarantees we can make regarding the planner's operation. Then we present an overview of the OS Manager; we defer a full discussion of this module till Section 3. As previously stated, we will attempt to stay at a high level, using examples to provide intuition.

2.1 The Goal Language

This section provides examples illustrating the expressiveness of our OS agent's goal language, and then explains how the OS agent interprets and satisfies user requests. Note that while we are concerned with *what* the user can say to the OS agent, we do not believe that the language we present here has the right *form*. Coating the language with syntactic sugar, and linking it to a graphical user interface, are topics of ongoing work.

The simplest request to the OS agent is to immediately respond to an external event. For instance, the following command instructs the agent to detect when a user named Neal becomes active on any machine by beeping.

```
(request (beep-twice ?self neal) :when (active.on neal ?machine))
```

Note that variable names begin with "?". Thus, `?machine` in the example means that we are leaving as unspecified the machine we expect Neal to be active on.

In general, such requests have two components: an action designation and a logical expression that denotes under what circumstances the action(s) should be executed:

```
(request <action>* :when <logical expr>)
```

The request need not employ a primitive action such as `beep-twice`. The triggered action may be a high-level goal in itself. Constraints are often universally quantified. For example:

```
(maintain (forall ?file (parent.directory ?file /papers/softbots)
                        (protection ?file group.readable)))
```

This request instructs the OS agent to make sure that *all* the files in the directory `/papers/softbots` are group readable.

Temporal constraints such as "satisfy once, at midnight" or "repeatedly, during the next ten days" can be associated with each request. The temporal designation is preceded by the keyword `:duration` or by the keyword `:until` and is followed by an arbitrary logical expression. Defaults are used when, as above, the temporal constraints are not specified explicitly. The default for standard requests is "do this once, now" and the default for `maintain` requests is "do this always." Thus, notice that in the example above, we ensure that all files, including files not in existence now, in the directory `/papers/softbots` are kept group-readable (c.f., the Unix command "`chmod g+r *`").

Below, we explain *how* the OS agent satisfies high-level user requests.

2.2 The Goal Manager

The Goal Manager (GM) is a relatively simple module that receives goals from the user and responds by issuing appropriate calls to the planner. Given a goal, like the ones shown earlier, the GM asks the planner to inform it when the logical expression in the `:when` component becomes true. (The planner's operation is described in Section 2.3 below.) Once the logical expression is satisfied (e.g., the user Neal is active on the machine called Machine-1), the GM invokes the planner with the goal of executing the action(s) specified in the request (e.g., initiating a talk session between the user and Neal).

Many goals have a trivial duration such as "once" or "always." However, if the goal has an explicit duration, the GM invokes the planner with the goal of informing it when the duration has expired. When the planner indicates that this is the case, the GM terminates processing of the goal.

The GM interprets goals of the form `(maintain P :duration <duration>)` simply as requests of the form: `(request (P) :when (not P) :duration <duration>)`. Once the request is satisfied, the GM automatically issues a new request to the planner. The GM terminates the `maintain` goal when it is notified that the `duration` expression no longer holds.

2.3 The Planner

The OS agent's planner maps user requests to appropriate sequences of OS commands. The inputs to the planner are logical models of the OS commands at its disposal, and a user request. The models are currently provided by the system manager. Although not yet incorporated into our implementation, there are well-known algorithms for automatically learning and refining classes of action models. Furthermore, sophisticated users can add their own models.

The planner's goal is based on the user's original request. For instance, the planner's goal may be to retrieve all Postscript files made available by a particular person from a remote site and print them locally. (In Section 2.4 we show how to specify this goal and the planning actions it generates). To satisfy this goal, the OS agent has to access the remote machine, retrieve the desired files, select an appropriate local printer, and print the files.

Given a goal, the planner dynamically synthesizes a sequence of OS commands that will satisfy the goal (this sequence is called a *plan*) and invokes the OS Manager to execute the plan. To dynamically generate a plan, the planner has to represent the available OS commands. The representation ought to answer (at least) two fundamental questions about each command:

- *Under what conditions will the command execute successfully?*
  The answer is a set of necessary conditions for including the command in a plan, which are referred to as *preconditions*. For instance, the preconditions for the command `lpr paper.ps` are that `paper.ps` exist in the current directory, that it be readable by the OS agent, and so on.
- *What is the effect of executing the command?*
  We refer to the effects of a command as its *postconditions*. The postcondition of `lpr` is simply to send the file to a printer. Other commands have multiple postconditions (e.g., `ftp` changes the OS agent's current directory, shell, and machine) or postconditions that provide the OS agent with information, rather than changing the system's state (e.g., `wc` tells the OS agent about a file's length).

An action model can be viewed as a generalization of a Prolog inference rule to allow for multiple postconditions, universal quantification, and state change. The precise syntax and semantics of our action representation language are described in [6], and sample action models appears in Figures 3 and 4. Note that while `wc` might appear to be a trivial example, recall that the planner might need to use that command as part of a larger plan. Hence the planner needs to know preconditions and postconditions. The example involving `netfind` is more realistic and is representative of the capabilities of the planner.

```
Name:  (WC ?file)
Preconditionss:                          Postconditions:
   (isa file.object ?file)                  (character.count ?file !char)
   (isa directory.object ?dir)              (word.count ?file !word)
   (name ?file ?name)                       (line.count ?file !line)
   (parent.directory ?file ?dir)
   (protection ?file readable)
```

Figure 3: Planner representation of the Unix command `wc` which provides the character, word, and line count of a file. The preconditions uniquely designate a file and ensure that the file is readable. The postconditions record the information gained by executing the command. The counts returned by `wc` are bound to the variables `!char, !word`, etc.

```
Name:      (netfind ?person)
Preconditions:                           Postconditions:
   (current.shell csh)                      (userid ?person !userid)
   (isa netfind.server ?server)             (person.machine ?person !machine)
   (firstname ?person ?firstname)
   (lastname ?person ?lastname)
   (or (person.city ?person ?keyword)
       (person.institution ?person ?keyword))
```

Figure 4: An action schema encoding the `netfind` utility; the planner uses it to determine when an internet service could help achieve a user goal. The disjunctive precondition encodes the fact that `netfind` requires the person's institution or city to constrain its search. The planner subgoals on obtaining that information from a distinct information source.

The basic planning process proceeds as follows. Based on the user's original request, the planner creates an initial goal. A planner goal is a quantified conjunction of atomic propositions, each of which is referred to as a *subgoal*. The planner maintains a data structure representing its plan. Initially, the plan contains a set of unsatisfied subgoals corresponding to the original input goal. To derive the plan, the planner repeatedly chooses a subgoal to satisfy and searches for an action with a postcondition that unifies with the subgoal. Once such an action is chosen, the planner inserts it into the plan, adding its preconditions to the list of subgoals "to be satisfied." The planner then starts another iteration of its cycle, choosing a new subgoal to satisfy, and so on. Some subgoals may be satisfied by the current system state, or by actions already inserted into the plan. The planner notices this by keeping a model of the system's state, checking whether a subgoal is already satisfied, and recording that fact. Planning can necessitate backtracking, i.e., the planner may

pursue a particular plan only to find that it contains a critical flaw. In this case, it backtracks through some of its earlier steps and tries another plan. For instance, it may plan to retrieve files from a particular machine and only then realize that its account on that machine has expired and that the machine does not support anonymous `ftp`. In essence, the planner carries out a backward-chaining search of the sort we might see in a Prolog engine. The planner is finished when it arrives at a plan where each action's preconditions are satisfied, and which brings the system to a state where the input goal is satisfied.

2.4 An Example

As an illustration, we describe an example of a plan that our current implementation of our OS Agent is capable of executing. [1] Suppose the agent is given the following goal: print all the files made available by Matt Ginsberg at his machine. This goal is specified as follows:

```
(forall ?file (author ?file ?person)                  subgoal 1
              (firstname ?person Matt)                subgoal 2
              (lastname ?person Ginsberg)             subgoal 3
              (person.machine ?person ?machine)       subgoal 4
        (printed ?file ?printer))
```

In order to satisfy this goal, the agent first has to satisfy four subgoals (numbered 1–4 above). It picks subgoal 4 first, because of a specific heuristic to localize the search in the planning process. In order to satisfy subgoal 4, the planner looks through action schemata, such as those shown in Figures 3 and 4, for action models whose postconditions match the subgoal. Based on this search, the agent decides to try the internet utility `netfind`. Subgoals 2 and 3 are used to constrain the preconditions for `netfind`, as shown in Figure 4. However, since `netfind` requires a person's institution to constrain its search, the agent plans to first obtain that information. In order to do this, the agent once again examines action schemata for action models whose postconditions match "`person.institution ?person ?keyword`". Based on this search, the agent picks a specific command that retrieves the institution information from the publication record from an INSPEC database.

The agent interleaves planning with execution. At this point, the agent executes a number of steps: accessing INSPEC; retrieving Ginsberg's institution; feeding that information to `netfind`; and retrieving his address, which is `t.uoregon.edu`. The agent's goal now is a universally quantified one of finding all of Matt Ginsberg's papers at a remote site. In order to satisfy this goal, the agent plans to access the remote site, enumerate all the files belonging to Ginsberg, and retrieve them. It therefore invokes `ftp`, `ls`, and `ftp get` commands to achieve this goal. (We omit some of the details here for brevity).

Thus the execution sequence for this simple plan is as follows:

- Try to satisfy the main goal by satisfying four suboals.
- Match fourth subgoal with `netfind`'s postconditions.

[1] In the example, we have made some changes to the goal specification and the omitted some of the details of the planning algorithm for conceptual simplicity

- Attempt to satisfy `netfind`'s preconditions by locating the person's institution.
- Apply the heuristic of looking up the list of published papers in the INSPEC database to locate an institution from it.
- Retrieve the institution name, and use it to satisfy `netfind`'s preconditions.
- Invoke `netfind` to locate the machine address for Ginsberg.
- Invoke `ftp` to this machine, and `cd` to `/pub`.
- Invoke `ls` to locate all files belonging to Ginsberg at this site.
- Invoke `get` to fetch each file.
- Invoke `lpr` to print each file.

There are several more subtle aspects to the planning process. First, actions that change the system's state can interact. For example, if the planner decides to execute `ftp`, it will put itself in the `ftp` subshell, and thus won't be able to execute commands such as `grep`, `wc`, etc. However, if the planner exits the `ftp` subshell it won't be able to do `ls` remotely, and so on. The planner has to order its actions appropriately to avoid harmful interactions and take advantage of useful ones. Second, due to the sheer size and dynamic nature of the system's state, the planner's model is necessarily partial or incomplete. As a result, information necessary for planning may be unknown to the planner (e.g., what is the protection on the file `paper.ps`).

Space precludes a comprehensive discussion of our planning algorithm (see [1, 3]). However, we would like to emphasize that planning is well understood as a search problem. Modern planning algorithms are provably (see [9]):

- **complete:** if a plan exists, the planner will find it, and
- **sound:** if the planner outputs a plan, that plan is guaranteed to achieve its goal.

These formal guarantees do not ensure that the planner is efficient. However, we have not found efficiency to be a problem in practice (see Section 4). The planner accepts control heuristics, specified in a high-level language, which constrain the planner's search by instructing it to ignore options, to prefer certain options over others, etc. These heuristics can be hand-coded or generated automatically via machine learning techniques [5, 8].

When the planner decides to execute a command, it sends a message to the OS Manager detailing the command and its arguments. The OS Manager responds with a message indicating whether the command was executed successfully and what information was obtained (e.g., a list of file names in the case of `ls`). If execution fails, the planner has the option of retrying the failed command at some later time, choosing an alternative command, or entering a debugging mode in which the planner attempts to determine the source of the error and to prevent it from recurring in the future. This flexibility enables the planner to choose the appropriate response to different types of execution failures.

System-dependent information is localized in the OS Manager, so the rest of the OS agent is invariant across different operating systems.[2] The results of execution may not be immediate. The agent may be

[2] Of course, the caveat to this statement is the planner's command models, which may vary in subtle ways from one system to another.

forced to wait for an *exogenous* event (i.e., an event outside of the agent's control) before acting. If so, the OS Manager assumes responsibility for detecting when the event occurs and immediately informing the planner about it. The OS Manager's internals are described in Section 3.

2.5 OS Manager

We have already described the OS agent's planning capability, which ultimately issues requests to the OS Manager. In this section we examine the OS Manager that supports those requests. At a high level, there are various classes of requests that the planner might make, each of which has its requirements; for example, the OS Manager must be able to supply information requested by the planner, it must be able to notify the planner of events that occur, usually asynchronously, and it must be able to manipulate operating system objects. Furthermore, it must be able to handle such requests in a distributed and heterogeneous environment.

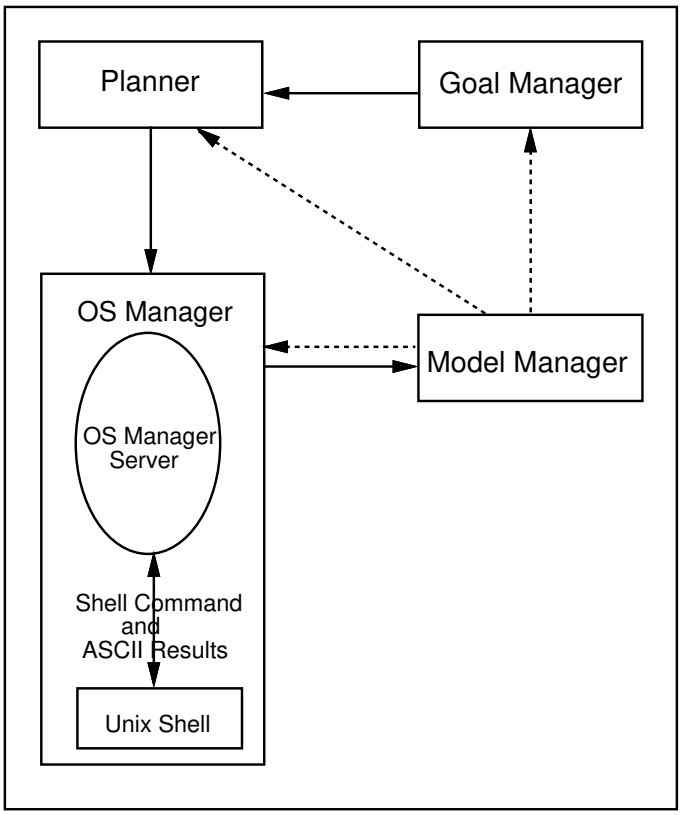

Figure 5: OS Agent with OS Manager Detail: In this structure, a dedicated OS Manager Server process invokes shell commands and parses the results of the ASCII strings returned by the shell.

The most obvious implementation, and in fact the one used by our first prototype of OS agents, was to rely completely on an existing operating system command interpreter (viz., a Unix shell). We show this pictorially in Figure 5. There are two problems with this approach. First, some information is not available through the command interface. Second, certain kinds of constraint or monitoring requests will require repeated polling at the command level, and over a network, this can result in unnecessary message

overhead, particularly when several OS agents are requesting the same information from one site. Although one can reduce the polling rate, this results in delayed notification. Thus, while a command interpreter can provide many of the functions needed by OS agents, sole reliance on a command interpreter is insufficient and potentially expensive in a distributed environment. We describe a more general and extensible design in Section 3.

3 System Support for OS agents

3.1 Prototype Server: Goals and Structure

The goals of our prototype implementation described here are twofold: (1) to provide a structure capable of handling distribution and heterogeneity, and (2) to provide support for extending the system to respond to new types of queries. It is this second objective that we describe in more detail in this section. We expect that as users become accustomed to a goal-oriented facility such as OS agents, they will think of new kinds of goals to express, some of which will require special operating system "hooks." Thus, we have designed a general-purpose mechanism to support the addition of such hooks to the operating system.

Recall that as a result of the planning function, the planner generates a set of requests for the OS Manager, which is then responsible for handling all interactions with the external world. In the original scheme, as shown previously in Figure 5, the OS Manager consists of a OS Manager Server that uses the facilities of the Unix shell. Figure 6 shows the detail of the OS Manager module in our new scheme. Here, we logically split the OS Manager into two components—an OS Manager clerk and an OS Manager server. Physically, the OS Manager clerk and OS Manager server are two separate processes, which in general will be located on different machines across a network. The clerk communicate with the server via remote procedure calls (RPCs). In turn, the OS agent server interacts with the operating system through standard system calls and commands where possible, and through extended mechanisms and abstractions (to be described in the next section) where necessary. In general, an OS Manager clerk will talk to multiple servers located on different machines using the RPC mechanism. The clerk executes on behalf of a particular user and is not trusted. It has only as much privileges as the user and thus, a malicious clerk cannot do any more harm to the system than an ordinary user.

By defining a clerk/server interface based on RPC, and locating in the server all of the operating system interface functions, we support both distribution and heterogeneity. We support distribution across the network, because a clerk can communicate with both local and remote servers in the same way; the RPC interface is identical in both cases. We support heterogeneity, i.e., the clerk and server can be running on different machine types and operating systems, because the RPC interface is standardized and machine independent. Conventional RPC stub generators, including ours, create specialized code called marshaling code to hide the details of converting between specific hardware details of the clerk and server machines.

3.2 Operating System Extensions for OS agents

This section presents the structure and the abstractions we have used to support the needs of OS agents within Unix. Our objective was to support general event signaling for OS agents, and to allow flexible system modification in order to provide additional information where needed. Such modification would be

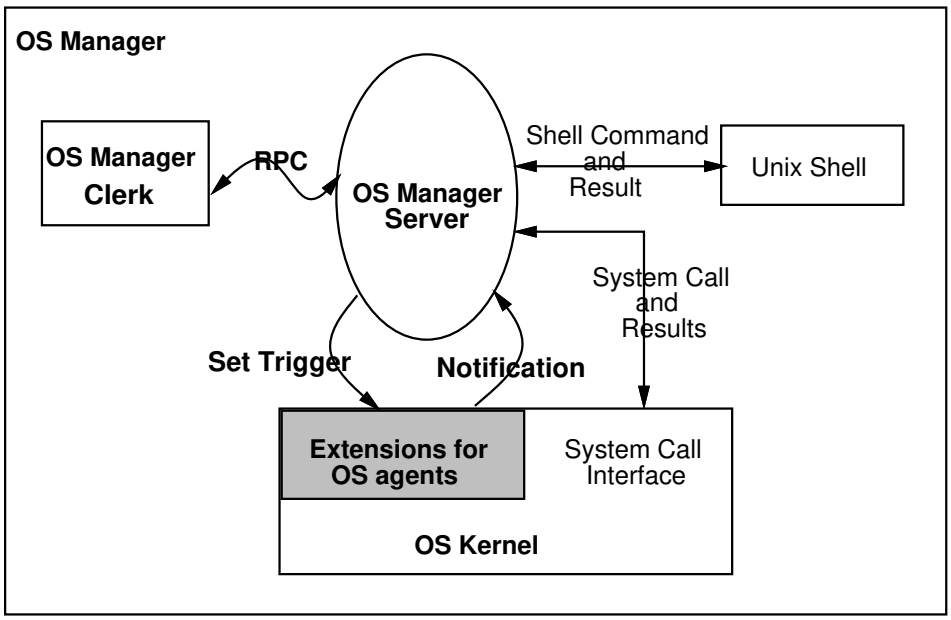

Figure 6: Structure of the OS Manager: This shows a detailed view of just the OS Manager. The OS Manager interacts with the Planner and the rest of the OS agent components as before.

permitted only to a trusted person, such as a system manager. Our approach has much in common with the approach used by performance monitors and debuggers; in fact, our mechanism could be used to support such facilities as well.

Event Specification

From the point of view of the operating system, an "event" usually signifies a change of state within the system, for example, a user enters a character, a file is modified, a kernel procedure is invoked, etc. We determine in our design, a priori, the *types* of events that we believe are useful for the purposes of an OS agent. Thus, "file modification" is an example of an event type. We expect the set of event types to change slowly over time. In contrast, event *instances* of interest, such as the modification of a *particular* file, change constantly.

In response to user requests, the OS Manager clerk makes RPC calls to appropriate OS Manager servers. Each server translates a clerk request into a set of event notification demands that the kernel is expected to fulfill. The server indicates its interest in a specific event instance by making a privileged system call that can only be made by trusted principals. We refer to this mechanisms as "setting a trigger" and indicate it as an arrow labeled *Set Trigger* in Figure 6. By setting a trigger, the operating system and the OS Manager server enter into a contract, whereby the system guarantees to notify the server each time that particular event instance occurs. Utilization of network and processor resources are consequently kept to modest levels, as both clerk and server threads block waiting for the kernel to signal the event.

OS Manager servers can limit the frequency of notification by specifying additional constraints that have to be satisfied before notification occurs. For example, rather than requesting notification each time a file is

modified, a server can confine its interest to cases where the file size grows beyond a specified threshold. We refer to these additional constraints as "predicates," which are boolean expressions that evaluate to true or false.

Though the current prototype does not implement this, our design allows the server to patch short code sequences into the kernel, to be executed when particular kernel events occur. Consider the following example of its use: an OS Manager server could cause code to be executed inside the kernel, perhaps incrementing a counter or setting a flag, each time a particular kernel procedure is invoked. The operating system mechanism needed to do this is straightforward and has been used in earlier systems to add performance probes or device drivers dynamically to a running kernel.

To summarize, event specification by OS agents is facilitated by three mechanisms. First, we define a relatively static set of event types. Second, OS agent servers, which are considered trusted, can dynamically express an interest in specific instances of these event types. The specific instances of events will depend on the requests that OS agent servers receive from their clerks. Third, we include a mechanism for patching small server-supplied code sequences into the kernel to signal specific event occurrences. Such code would be supplied by a trusted system person: it is a relatively heavyweight operation, so we do not expect new pieces of code to be installed very frequently.

3.3 An Example Implementation

To test the feasibility of our ideas, we modified a DEC Ultrix kernel to provide support for an OS Manager server. The framework presented in the previous section is sufficiently general to accommodate complex requests from OS Manager clerks. However, most current clerk requests can be satisfied by a subset of the functionality outlined above. Currently, our implementation only provides support for setting triggers on specific "i-nodes"; this allows us to be notified on logins, for example, by watching a specific system file modified by login. Thus, one way of viewing the current event notification facility is as a generalization of "watchdogs" [2], an earlier system permitting watchpoints on files.

In our implementation, the OS Manager server runs as a privileged process that receives RPC requests from a remote OS Manager clerk. The clerk's RPC request is blocked from proceeding until the server determines that the request is complete. A clerk request translates into a set of constraints or predicates that need to be satisfied; based on those predicates, the server issues commands or arranges if necessary for kernel event notification.

Kernel event notifications typically involve simple constraints, for example, with respect to the modification or access time of i-nodes. Depending on the complexity of the request, the server may need to make further queries after notification to ensure that additional constraints are satisfied. We rely on the server to ensure complex constraint satisfaction, rather than placing complex constraints in the kernel. This reduces the code executed in the kernel, and often a single event with a simple predicate meets the partial requirements of multiple user requests. In practice, a large number of OS agent requests can be readily handled with our current implementation.

There are situations, however, where our implementation might affect performance. For example, on a DEC Ultrix system with no quotas in effect, it is difficult to enforce file size limits; every file write call could potentially exceed the disk limit, thus the OS agent server could be repeatedly awakened on false alarms. Again, this is tied to the decision to evaluate complex predicates in the server. Therefore, for efficiency

reasons, some predicates such as this one are better evaluated in the kernel, at the cost of some additional kernel code complexity.

The Ultrix OS agent is an ongoing effort and additional triggers will be added as it matures. In the interest of expedience, our implementation makes some simplifications, and we have not yet invested any time in performance tuning. However, our experiments are real enough that we believe in the general structure, and we feel that extending the functionality and implementing it on other platforms would pose no major hurdles.

4 Performance Analysis

Now that we have seen the basic planning cycle of the OS agent, this section presents some measurements to show the time involved in satisfying some example requests. Our objective is simply to indicate that a mechanism such as OS agents is practical; obviously users would not be satisfied with a powerful but slow agent that takes 5 minutes to accomplish a task that the user can do in 2 seconds today.

Following are measurements from our prototype. As noted in the previous section, no effort has been made to optimize our clerk. Furthermore, the AI components of the agent consist of approximately 15K lines of Lisp code. We could easily achieve substantial speedup by re-coding the AI portions of the system in C.

Table 4 shows the time required by the Ultrix agent to satisfy a set of sample user goals. The agent is running on a lightly loaded 25 MHz DECstation 5000/200. The times are broken down into Planner time and Clerk time, where the Clerk time is further subdivided into time spent issuing system calls and time spent on event notification. The first goal simply translates to an `uptime` command, executed remotely on `machine-1`; the second goal requires the agent to identify Draper's userid and home machine utilizing the `netfind` facility, the local `staffdir` database, or `finger`, then beep when Draper becomes active on her machine. The `maintain` goal requires the agent to detect that the disk utilization exceeds 75%, and to respond by compressing Postscript files, deleting old backup files, and moving certain directories to a different disk. We report the time so as to detect that the disk utilization is too high and to rectify the problem. Finally, the agent registers its interest in the file TR-93-03-22 at a remote site. When the clerk detects that the TR is available, the agent retrieves the file with anonymous `ftp`, prints it locally, and monitors the print job to ensure completion. We report the time from TR availability to print initiation.

Agent Goal	Planner time	Clerk time		Total
		system calls	notification	
`(find-load m-1 ?load)`	0.51	1.29	-	1.80
`(contact alicen draper)`	1.37	1.83	0.01	3.21
`(maintain (disk.util.below disk-1 75))`	1.82	2.71	0.01	4.54
`(retrieve TR-93-03-22)`	2.52	3.19	0.05	5.76

Table 1: Sample Goal Satisfaction Times (in seconds of real time); the numbers illustrate that, even in our unoptimized prototype, OS agents do not impose an exorbitant performance penalty on the user.

The last column shows that, although invoking the agent is clearly slower than directly executing the equivalent shell commands (as show in the second column), the performance penalty is modest. The table suggests that our implemented agent's performance is reasonable, as it stands, but how will its running time scale with various parameters, such as the number of commands known to the agent, the number of event types, the number of "active" event instances (or triggers), and so on?

We anticipate that the number of event types will not increase significantly, but the number of triggers could become large during some periods. To test the effect of kernel triggers, we experimentally loaded the system with a number of background processes, each of which sets a trigger and is periodically notified by the kernel. We then examined the response time effect, both to a process making direct queries to the OS agent server, and to several user-level commands. The query process saw only a 300 μs decrease in response time to its server requests when the system was loaded with 50 processes and triggers. Our user requests saw no change in elapsed time (as reported to the nearest second by the `time` command) at that level, but with 64 trigger processes, `time` reported a 1 second increase in response time for a LaTeXof this paper. We believe as a result of these basic tests that the performance impact of triggers is not significant.

As the agent increases in sophistication, the number of predicates, commands, and objects (e.g., files, hosts, users) it knows will increase substantially. However, this does not necessarily imply that the agent will slow down. Recall that to satisfy a user goal, the agent searches the space of plans (i.e., legal combinations of known commands) in a directed fashion. To satisfy each goal, the agent considers all the operators whose postconditions unify with that particular goal. If we bound the number of such operators by b, and bound plan length by d, then planning time is $O(b^d)$. In many cases, the number of known commands can increase substantially without increasing b or d. While some goals, such as locating a user's e-mail address, can require trying many alternatives, many other goals do not: `cd` is the operator for changing directories, `mv` is the operator renaming files, etc. Search-control heuristics also serve to reduce b, often guiding the agent to the small set of commands likely to satisfy its goal. Ultimately, the agent's planning speed depends on the amount and quality of search-control heuristics at its disposal. If the agent is provided with (or is able to automatically learn) adequate control knowledge, we can expect the performance to degrade gracefully as the agent increases in sophistication and knowledge.

5 Conclusions

We have described the philosophy, design, and implementation for OS Agents, a goal-oriented operating system command mechanism that uses AI planning techniques to satisfy its objectives. With OS agents, a user can request that the system carry out a complex action, and the system will automatically determine the steps needed to carry out that request. Since planning is done dynamically, the agent is capable of responding to changes in the system's state and configuration. Our current implementation is based on Unix, but in the framework of a distributed and potentially heterogeneous environment.

The agent metaphor has become popular recently, however OS agents differ sharply from this body of work in several ways. We have successfully incorporated well-understood AI planning algorithms into our agent, yielding a flexible and extensible system. The precise descriptions of the planning algorithm and operator representation language in [6, 9], combined with the discussion in this paper, suffice to replicate our implementation and experimental results. In addition, the OS agent's normal operation is rife with learning opportunities. Algorithms already exist for learning control heuristics by analyzing past successes

and failures [5, 8], and automatically generating logical models of actions based on experiments and by observing human users [4, 7, 10]. In future work, we plan to incorporate these algorithms into the OS agent and investigate their performance in the Unix domain.

We have also described the integration of the OS agent facility within Unix. An OS agent server is responsible for interfacing to the system, issuing commands, invoking system calls, and setting triggers on internal system events of interest. We have provided some privileged support within Unix so that the agent can express interest in particular system states without requiring polling or privileged access. The mechanism also allows a trusted person to dynamically add additional event types to the system in support of OS agent queries.

We believe that a facility such as OS agents will permit users to issue increasingly powerful requests. As experience is gained, users will imagine new types of queries and requests that may require information or event signaling not originally envisaged by the system designers. For this reason, designers of future operating systems intended to support such a facility should think seriously about the information and event signaling needs, and should anticipate the need for system extension in support of user requests.

5.1 Current Status

At the time of writing, an implementation of the ideas presented here has been running for over a year at the University of Washington. We do not normally use the OS extensions because they require modified kernels to be installed on our machines. As well, we have designed various graphical user interface packages to provide syntactic sugar for the goal specification language.

Acknowledgments

The authors would like to thank the groups at INRIA project SOR and Bull/IMAG for their support. Thanks are also due to Ellen Siegel and Solange Karsenty for commenting on earlier drafts of the paper which greatly added to its clarity.

References

[1] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, August 1990.

[2] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX file system. In *Proceedings of the 1988 Winter USENIX Conference*, pages 267–275, February 1988.

[3] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, July 1987.

[4] T. G. Dietterich. *constraint propagation techniques for theory-driven data interpretation*. PhD thesis, Stanford University, 1984.

[5] O. Etzioni. STATIC: A problem-space compiler for Prodigy. In *Proceedings of AAAI-91*, 1991.

[6] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proceedings of KR-92*, October 1992.

[7] Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1992. Also appeared as Technical Report CMU-CS-92-175.

[8] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3), March 1990.

[9] J. Penberthy and D. Weld. UCPOP: a sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, pages 103–114, October 1992.

[10] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. In *Proceedings of FOCS-87*, October 1987.

[11] M. F. Schwartz and P. G. Tsirigotis. Experience with a semantically cognizant Internet white pages directory tool. *Journal of Internetworking: Research and Experience*, 2(1):23–50, March 1991.