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
April 12, 1993

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 FR-35
University of Washington
Seattle, WA 98195

April 12, 1993

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 and pipes, 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.

1 Introduction

The expressive power of operating system command interfaces, whether textual or visual, has evolved little over the lifetime of operating systems. In this paper we present a new *goal-oriented* approach to the

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. C. Thekkath is supported in part by a fellowship from Intel Corporation. H. Levy is supported in part by a Fulbright research award and by INRIA.

operating system command interface. Our approach enables the user to request tasks that while simple to specify, may be complex to carry out with conventional commands. Often such requests involve monitoring status or maintaining constraints. For example, the following list presents classes of reasonable user requests that are difficult to express using a conventional (e.g., Unix-like) command interface:

1. **monitoring events:**
   - Send me mail if my disk utilization exceeds eighty percent.
   - Tell me when Neal logs into his workstation.
   - Immediately display any posts containing the string "bicycle" that appear on the market bulletin board this week.
2. **enforcing constraints:**
   - Keep all files in the directory `/papers/sosp` group-readable.
   - Ensure that all my Postscript files are current (i.e., automatically generate a new postscript file whenever the corresponding TEX file is modified).
3. **locating and manipulating objects:**
   - At midnight, compress all of my files that have not been accessed in a week whose size exceeds 10 megabytes.
   - Print my file on any non-busy printer on the 4th or 5th floor, and tell me where to find it when the print is done.

These task classes are neither exhaustive nor mutually exclusive, but illustrate our main point: an ideal interface should enable 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 OS interfaces are incapable of such expressiveness.

This paper describes the goals, design, and implementation of *OS agents*, which satisfy high-level user requests, as well as the operating system implications for support of OS agents. 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. This approach has a number of obvious advantages over the typical command/programming approach. First, the system is 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). Second, if one method for accomplishing the task fails unexpectedly, the system can fluidly recover and try a different method. Third, the language for specifying goals to the OS agent is system independent, making evolution or even radical change of the system transparent to the user.

OS agents may be viewed as a command-language extension mechanism, as are shell scripts [6]. However, to match the power of OS agents with shell scripts or conventional programs, 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 contrast, once the OS agent knows about a new facility, that facility becomes immediately available to its planning process, and is automatically invoked to satisfy relevant user requests.

To achieve this functionality, an OS agent must be able to perform the following:

- *Dynamically synthesize a sequence of OS commands, system calls, or program invocations 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.

- *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 that meets these needs may sound difficult, in fact, the base technology already exists within two major areas of Artificial Intelligence (AI) known as planning and machine learning [1, 21]. In recent years, concise algorithmic descriptions and public-domain implementations have become available, attesting to the maturity of the techniques [3, 18].

In this work, we are demonstrating the marriage of what may seem like strange bedfellows — AI and operating systems — in order to add expressive power to the user's command interface. The organization of this 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 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. We conclude Section 2 with a detailed comparison between the OS agent approach and the traditional shell script or programming approach to command language extension.

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, both through commands and system calls, our goal was to prototype a general, extensible service on which OS agents could call for their needs.

Our implementation considers both distribution and heterogeneity; while the current implementation and examples are in Unix, we are not depending on its specific properties. It is straightforward to link the agents to other operating systems such as Mach, VMS, NT/Windows, etc.

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 **Planner** synthesizes sequences of OS commands to satisfy requests received from the user. Commands to be executed are sent to the OS agent clerk.
- The **Clerk** handles all interaction between the OS agent and its external environment, including inter-node communication (via RPC), command execution, status checking, and so on. The design and implementation of the clerk is described in Section 3.

In the first subsection below, we present examples of the goal language to indicate how a user could specify goals. Then, we present the planner's algorithm and explain the formal guarantees we can make regarding the planner's operation.

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 for future 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, and initiate a talk session with him.[1]

```
(request (talk ?self neal) :when (active.on neal ?machine))
```

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 `talk`. The triggered action may be a high-level goal in itself:

```
(request (disk.util.below disk-1 70%)
         :when (not (disk.util.below disk-1 80%)))
```

The agent is thus instructed to reduce the disk utilization via all the means at its disposal including compressing files, moving them to under-utilized disks, sending irate messages to system users, etc. In many cases, such requests are conveniently specified as constraints:

[1] Variable names begin with "?" as in `?self` or `?machine`.

```
(maintain (disk.util.below disk-1 75%))
```

Constraints are often universally quantified. For example:

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

This request instructs the OS agent to make sure that *all* the files in the directory `/papers/sosp` are group readable. Finally, note that temporal constraints such as "satisfy once, at midnight" or "repeatedly, during the next ten days" can be associated with each request. 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." Below, we explain *how* the OS agent satisfies high-level user requests.

2.2 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, algorithms exist for automatically learning and refining classes of action models (e.g., [7, 12, 19]). Furthermore, sophisticated users can add their own models.[2]

The planner's goal is based on the user's original request. Planner goals are quantified conjunctions of atomic propositions. Each atomic proposition in the goal is referred to as a *subgoal*. For instance, the planner's goal may be to retrieve all 1993 tech reports from a remote site and print them locally, which would be expressed as follows:

```
(forall ?file (machine ?file cs.stanford.edu)
              (parent.directory ?file /pub/trs)
              (creation-date ?file ?date)
              (year ?date 1993)
        (printed ?file ?printer))
```

To satisfy this goal, the OS agent has to access the remote machine, retrieve the desired files, reformat them, select the appropriate local printers, 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 clerk 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

[2]We expect both users and managers at different sites to share command models, so we do not expect the one-time burden of encoding OS commands in this language to be onerous.

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 [10], and a sample action model appears in Figure 1.

```
Name:  (WC ?file)
Preconds:                              Postconds:
   (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 1: 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.

The basic planning process proceeds as follows. 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. The planner may pursue a particular plan only to find that it contains a critical flaw. 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.

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, 5]). However, we would like to emphasize that planning is well understood as a search problem. Modern planning algorithms are provably (see [18]):

- **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 [9, 17].

When the planner decides to execute a command, it sends a message to its clerk detailing the command and its arguments. The clerk 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 clerk, so the rest of the OS agent is invariant across different operating systems.[3] The results of execution may not be immediate. The agent may be forced to wait for an *exogenous* event (i.e., an event outside of the agent's control) before acting. If so, the clerk assumes responsibility for detecting when the event occurs and immediately informing the planner about it. The clerk's internals are described in Section 3.

2.3 OS agents versus Standard Utilities

It is instructive to compare OS agents with more standard methods of system extension, namely shell scripts and shell languages. Although a competent system programmer could write an application or shell program to satisfy many of the *individual* requests we have listed, OS agents have a number of advantages:

- 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.

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

- 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 (e.g., printer-3 is jammed).

- 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 applications. 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 [9, 17].

- 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.

- Finally, in the absence of the event notification services provided by our clerk, shell programs are reduced to frequent and expensive polling (or infrequent but unreliable polling) to notify the user about events of interest.

To illustrate the extensibility of our approach, consider the goal of finding a user's e-mail address given his name (a common goal 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 [20]. 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 database, no programming; yet, `netfind` can now be used as part of the plan to satisfy any goal. Furthermore, given the BNF of our command-model language, and definitions for the predicates we use, programmers can encode models of their applications and make the applications directly available to the OS agent.

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. 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.

3 System Support for OS agents

The previous section described the OS agent's planning capability, which ultimately issues requests to the operating system. In this section we examine the operating system server 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 server 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.

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). 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 implementation below.

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.

Figure 2 shows the high level organization of the system, most of which has been previously described. As a result of the planning function, the planner generates a set of requests for the clerk, using standard procedure calls. The clerk is then responsible for handling all interactions with the external world, which is done through RPC calls to OS agent *servers* on its node and on other nodes. In turn, the OS agent servers interact with the operating system through standard system calls and commands where possible, and through extended mechanisms and abstractions where necessary. 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: distribution because a clerk can communicate with both local and remote servers in the same way; heterogeneity because the RPC interface is standardized and machine independent.

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 signalling for OS agents, and to allow flexible system modification in order to provide additional information where needed. Such modification would be

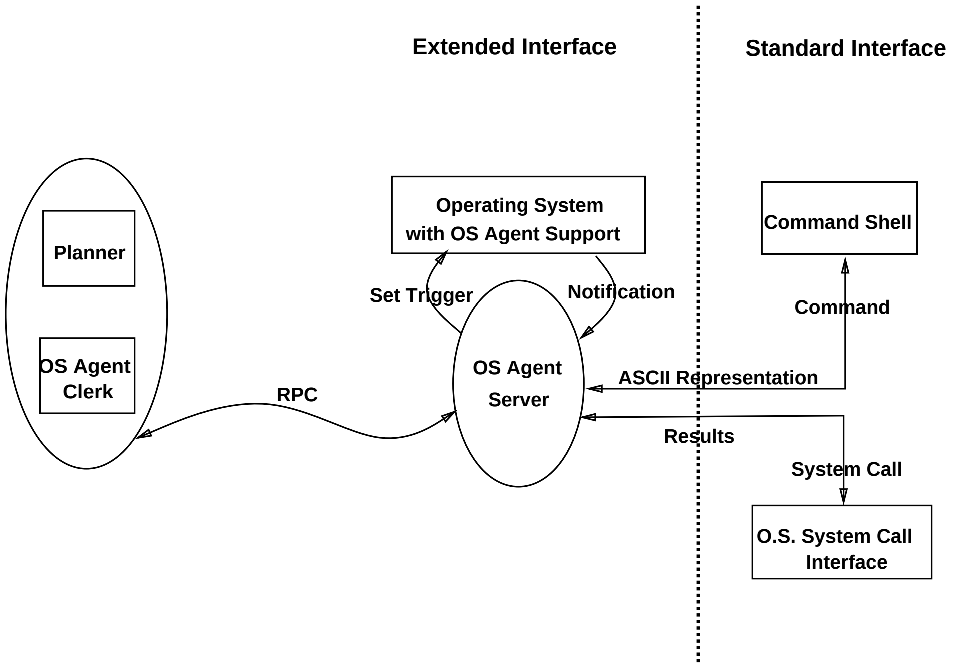

Figure 2: Structure of the OS agent

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 [16, 4]; 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 agent clerk makes RPC calls to appropriate OS agent 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 2. By setting a trigger, the operating system and the OS agent 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 agent 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 agent 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 the past to add performance probes [11] or device drivers [22] 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 agent server. The framework presented in the previous section is sufficiently general to accommodate complex requests from OS agent 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 agent server runs as a privileged process that receives RPC requests from a remote OS agent 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 and the implementation of its server, 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 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.

Although invoking the agent is clearly slower than directly executing the equivalent shell commands, 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 Latex of 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 remain nearly constant 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 [8, 13, 15, 14], 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 as discussed in Section 2.3. The precise descriptions of the planning algorithm and operator representation language in [10, 18], 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 [9, 17], and automatically generating logical models of actions based on experiments and by observing human users [7, 12, 19]. 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.

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] W. Buntine and R. Caruana. Introduction to IND and recursive partitioning. NASA Ames Research Center, Mail Stop 269-2 Moffet Field, CA 94035, September 1991.

[4] D. J. Campbell and W. J. Heffner. Measurement and analysis of large operating systems during system development. In *Proceedings of the 1968 Fall Joint Computer Conference*, pages 903–914, December 1968.

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

[6] Computer Systems Research Group, Department of EECS, University of California, Berkeley, CA 94720. *CSH(1) UNIX User's Reference Manual (URM) 4.3 BSD*, 1986.

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

[8] R. Droms. Access to Heterogeneous Directory Services. In *IEEE INFOCOM '90*, pages 1054–1061, San Francisco, CA, June 1990.

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

[10] 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.

[11] D. Ferraari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1983.

[12] 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.

[13] R. E. Kahn and V. G. Cerf. An open architecture for a digital library system and a plan for its development. Technical report, Corporation for National Research Initiatives, March 1988.

[14] B. Laurel, T. Oren, and A. Don. Issues in multimedia interface design: Media integration and interface agents. In *CHI '90 Conference Proceedings*, pages 133–139, 1990.

[15] P. Maes and R. Kozierok. Learning interface agents. In *Proceedings of INTERCHI-93*, 1993.

[16] G. Mcdaniel. METRIC: A kernel instrumentation system for distributed environments. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 93–99, November 1977.

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

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

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

[20] 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.

[21] J. Shavlik and T. Dietterich, editors. *Readings in Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1990.

[22] Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View CA 94043. *Writing SBus Device Drivers*, 1991.