From: AAAI Technical Report SS-94-03. Compilation copyright © 1994, AAAI (www.aaai.org). All rights reserved.

Building Softbots for UNIX (Preliminary Report)*

Oren Etzioni **Neal Lesh** **Richard Segal**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{etzioni, neal, segal}@cs.washington.edu

Abstract

AI is moving away from "toy tasks" such as block stacking towards real-world problems. This trend is positive, but the amount of preliminary groundwork required to tackle a real-world task can be staggering, particularly when developing an integrated agent architecture. To address this problem, we advocate real-world software environments, such as operating systems or databases, as domains for agent research. The cost, effort, and expertise required to develop and experiment with software agents is relatively low. Furthermore, software environments circumvent many thorny, but peripheral, research issues that are inescapable in other environments. Thus, software environments enable us to test agents in a real world yet focus on core AI research issues. To support this claim, we describe our project to develop UNIX[1] *softbots* (*soft*ware ro*bots*)—complete intelligent agents that interact with UNIX. Our fully-implemented softbot is able to accept a diverse set of high-level goals, generate and execute plans to achieve these goals in real time, and recover from errors when necessary.

Motivation

The work described in this report is motivated by two fundamental claims:

- **Real-world software environments are attractive testbeds for AI research.** Specifically, environments such as operating systems, databases, and computer networks have the following features:
  - *pragmatic convenience:* the cost, effort, and expertise necessary to develop and experiment with software artifacts are relatively low. Software also facilitates the dissemination and replication of research results.
  - *realism and richness:* in contrast to simulated physical environments, software environments are real, providing a rich source of intuitions, motivating examples, simplifying assumptions, stumbling blocks, test cases, etc.
  - *research focus:* software environments circumvent many thorny research issues (e.g., overcoming sensory noise, representing liquids, shapes, etc.), enabling us to focus on core AI problems.
- **AI is mature enough to yield useful software agents.** Two examples of mature AI techniques are:
  - *Planning:* endowing a software agent with planning, execution, and error recovery capabilities will enable the user to specify high-level goals and expect its software agent to figure out how to best satisfy the given goals.
  - *Machine learning:* learning capabilities will enable a software agent to customize itself to a user, adapt to a changing environment, discover new resources (e.g., new bulletin boards, databases, etc.), and more.

The above claims are empirical. We set out to refine and validate them by developing *softbots* (*soft*ware ro*bots*) for UNIX. Our twin goals are to make fundamental contributions to core AI, relying on UNIX as a testbed, and to develop technology that will assist actual UNIX users. The two goals are synergistic. Our research goal leads us to ambitious softbot designs; our technological goal forces us to be practical and to focus on developing useful softbot capabilities.

The remainder of this report is organized as follows. Below, we describe the notion of a softbot in more

*This is an abridged and slightly revised version of a report we began to circulate in November 1992, describing our first fully-implemented softbot. Since that time, we have incorporated a more powerful planner into the softbot [Etzioni *et al.*, 1994, Golden *et al.*, 1994], broadened our focus from UNIX to the Internet, developed a graphical user interface to the softbot, and formulated the First Law of Softbotics [Etzioni and Weld, 1994]. In addition, a number of people have joined the project including Greg Fichtenholtz, Terrance Goan, Keith Golden, Mike Perkowitz, Rob Spiger, and Dan Weld. However, the softbot's architecture (Figure 1), and the spirit of our investigation, remain the same. Our research was supported in part by Office of Naval Research Grant 92-J-1946, and by National Science Foundation Grants IRI-9211045 and IRI-9357772. Richard Segal is supported, in part, by a GTE fellowship.

[1]UNIX is a trademark of AT&T Bell Labs.

detail. The follow section presents Rodney, a general-purpose UNIX softbot that we are continuing to extend and improve. We conclude with a discussion of future work.

Softbots

A *softbot* is an agent that interacts with a software environment by issuing commands and interpreting the environment's feedback. A softbot's effectors are commands (e.g. UNIX shell commands such as `mv` or `compress`) meant to change the external environment's state. A softbot's sensors are commands (e.g. `pwd` or `ls` in UNIX) meant to provide the softbot with information about the environment. Due to the dynamic nature and sheer size of real-world software environments it is impossible to provide the softbot with a complete and correct model of its environment; sensing and learning are essential.

Some will argue that programs such as computer viruses or even shell scripts are degenerate cases of softbots. Below we list and describe several properties that distinguish our softbot programs from shell scripts and viruses. The extent to which to which a program possesses the following capabilities is the extent to which it is *usefully* described as a softbot.

1. **Goal-directed behavior** —a softbot attempts to achieve explicit goals. Unlike a "Brooksian Creature," it does not merely follow pre-programmed instincts. Thus, a human can "program" a softbot by specifying goals for it.
2. **Planning, executing, and error recovery** — A softbot is able to compose the actions it knows about into sequences that, once executed, will achieve its goals. Furthermore, a softbot actually executes such sequences, monitors their progress, and attempts to recover from any unanticipated failures.
3. **Declarative knowledge representation** — A softbot stores its knowledge declaratively, enabling it to use the same knowledge in multiple tasks.
4. **Learning and adaptation** — A softbot improves its performance over time by recording and generalizing from its experiences. For example, we would like our UNIX softbots to learn new commands, the locations of various objects (e.g. specific files) and the preferences of its human partners.
5. **Continuous operation** — A softbot should continuously operate in its environment. This constraint forces softbot designers to address problems such as surviving, co-existing with other agents, and being productive over time.
6. **Natural-language processing** — Much of the information potentially available to a softbot is encoded in natural-language (e.g. UNIX man pages). A softbot's ability to understand its sensory inputs scales with its ability to understand natural language. Thus, a softbot requires strong natural-language capabilities.
7. **Communication and cooperation** — Many other agents (both human and softbotic) may be operating in the softbot's environment. The softbot's ability to achieve its goals may well depend on its ability to communicate and cooperate with other agents, including other softbots.
8. **Mobility and cloning** — Unlike most software, a softbot is mobile. For example, a UNIX softbot is able to move from its home machine to a remote machine by logging into a the new machine, copying itself, and starting a Lisp process on the remote machine. Thus, unlike hardware artifacts, a softbot can clone itself. [2]

Naturally, the softbots we have constructed to date possess only a small (but growing!) subset of the above capabilities. We now describe our softbot in more detail.

Rodney: A General-Purpose UNIX Softbot

This section presents Rodney, a general-purpose UNIX softbot. Rodney accomplishes tasks that fall into the following broad categories:

- **monitoring events:** immediately display on my screen any post that appears on the market bulletin board and contains the string "bicycle."
- **enforcing constraints:** keep all the files in the directory `/joint-paper` group-readable.
- **locating and manipulating objects:** at midnight, compress all files whose size exceeds 10 megabytes and have not been modified in more than a week.

We envision, but have not yet implemented, more sophisticated tasks including mail filtering, library and database searches, intelligent intrusion detection, etc.

Below, we describe Rodney's architecture and provide examples of Rodney "in action." Rodney's architecture is shown in Figure 1. The architecture has four major components:

- **Goal manager:** receives task specifications from the user, and periodically invokes the planner.
- **Planner:** satisfies goals by interleaving planning and execution. The planner is discussed in more detail below.
- **Model manager:** serves as the central repository for Rodney's beliefs. The model manager stores beliefs about the world's state, Rodney's operator models, etc.

[2] The genetic algorithms community has studied the evolution of simple artifacts such as bit strings or rules, but we would like to clone whole softbots.

- **Executor:** issues commands to the UNIX shell and interprets the shell's output. All direct interaction between Rodney and its external environment takes place here.

This report focuses on Rodney's planner and goal manager. Rodney's executor and model manager are relatively straight forward.

Planning with Incomplete Information

The first problem facing Rodney is representing UNIX shell commands as operators it can plan with. It is natural to think of certain UNIX commands such as `mv`, `cd` or `lpr` as operators, and of some UNIX tasks as goals (e.g., `(protection file1 readable)`) in a classical planning framework. However, UNIX has a number of more challenging aspects as well:

- Due to the vast size of the UNIX environment, any agent's world model is necessarily incomplete. For instance, no agent knows the names of all the files on all the machines accessible through the Internet.
- Due to the dynamic nature of the environment, beliefs stored by the agent frequently become out of date. Users continually log in and out, files and directories are renamed or deleted, new hosts are connected etc.

Consequently, many of the most routine UNIX commands (e.g., `ls`, `pwd`, `finger`, `lpq`, `grep`) are used to gather information, and Rodney has to represent information-gathering actions, and to confront the problem of planning with incomplete information.

Highly-expressive logical axiomatizations of the notions of knowledge and belief have not yielded planning algorithms or implemented planners. To facilitate planning in the UNIX domain, we chose to devise the less expressive but more tractable UWL representation. The syntax and semantics of UWL are specified precisely in [Etzioni *et al.*, 1992]. We provide a brief, intuitive discussion of the language below.

Our first step is to extend the truth values a proposition can take on: propositions can be either true `T`, false `F`, or "unknown" `U`. Truth values of `U` apply to propositions about which the planner has incomplete information: those that are not mentioned in the initial state and which no subsequent plan step has changed. UWL denotes propositions by content, truth value pairs such as `((current.directory rodney dir1) . t)`.

Next, UWL divides an operator's effects into those that change the world (`cause` postconditions), and those that change the planner's state of information (`observe` postconditions). Causal postconditions correspond to STRIPS' adds and deletes. Observational postconditions come in two forms, corresponding to the two ways the planner can gather information about the world at run time: it can observe the truth value of a proposition, `(observe ((P c) . !v))`, or it can identify an object that has a particular property, `(observe ((P !x) . t))`. The variables `!v` and `!x` are *run-time* variables, which refer to information that will not be available until execution time. For example, the UNIX command `wc` has the postcondition: `(observe ((character.count ?file !char) . t))`, indicating that the value of `!char` will only be available after `wc` is executed.

Finally, UWL annotates subgoals with `satisfy`, `hands-off`, or `find-out`. A `satisfy` subgoal can be achieved by any means, causal or observational. The subgoal `(find-out (P . t))` means roughly that the Rodney ought to determine that `P` is true, without changing `P`'s state in the process. Typically, Rodney ensures this by achieving `find-out` goals with observational postconditions. This constraint is critical to information gathering. Otherwise, if we give Rodney the goal `(find-out ((name ?file core-dump) . t))`, it may satisfy it by renaming the file `paper.tex` to be `core-dump`, which is not the desired behavior (and will have disastrous consequences if we then delete the file named `core-dump`). In general, if a planner is given a definite description that is intended to identify a *particular* object, then changing the world so that another object meets that description is a mistake. The appropriate behavior is to scan the world, leaving the relevant properties of the objects unchanged until the desired object is found. Subgoals of the form `(hands-off (P . t))` explicitly demand that the plan do nothing to change `(P . t)`'s state. For instance, we may use a `hands-off` constraint to prevent Rodney from deleting any files:

```
(hands-off ((isa file.object ?file) . F))
```

The UNIX command `wc`, represented as a UWL operator, appears in Figure 2. Other examples appear later. To date, we have represented more than forty UNIX commands in UWL, and are in the process of encoding many more.

```
Name: (WC ?file)
Preconds:
   (find-out ((isa file.object ?file) . t))
   (find-out ((isa directory.object ?dir) . t))
   (find-out ((name ?file ?name) . t))
   (find-out ((parent.directory ?file ?dir) . t))
   (satisfy ((protection ?file readable) . t))
   (satisfy ((current.directory softbot ?dir) . t))
Postconds:
   (observe ((character.count ?file !char) . t))
   (observe ((word.count ?file !word) . t))
   (observe ((line.count ?file !line) . t))
```

Figure 2: UWL model of the UNIX command `wc`.

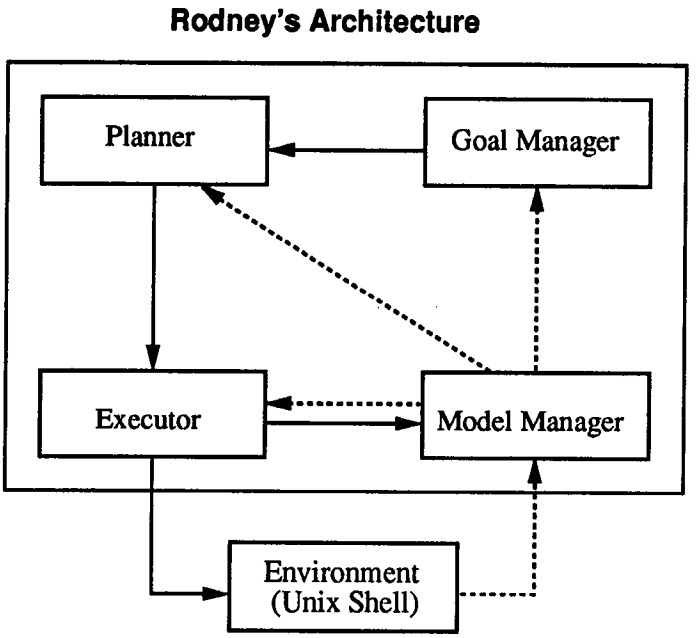

Figure 1: A high-level view of Rodney's Architecture. Loosely speaking, the solid arrows represent control flow in Rodney, and the dashed arrows indicate data flow.

Rodney's Planner

Rodney's planner attempts to satisfy UWL goals received from the goal manager. Classical planners (e.g. [Chapman, 1987, Fikes and Nilsson, 1971]) presuppose correct and complete information about the world. Rodney has incomplete information about its UNIX environment. As [Olawsky and Gini, 1990] point out, a planner with incomplete information has three basic options: to derive conditional plans that are intended to work in all contingencies, to make assumptions regarding unknown facts, and replan if these assumptions turn out to be false at execution time, or to interleave information gathering with planning to obtain the information necessary to complete the planning process. Conditional planning is difficult to pursue when many of the contingencies are not known in advance.[3] Making assumptions is appropriate in some cases (e.g., we may assume that a file is appropriately protected, and replan if we get an execution error), but not in others. For instance, if we are asked to locate a file with a certain name, it seems inappropriate to *assume* the answer. In Rodney, we have chosen to interleave planning and information gathering.

Rodney's planning algorithm is based on that of SNLP [McAllester and Rosenblitt, 1991]. We review SNLP briefly, and then describe our extension. The central SNLP data structure is a plan representation, which contains goals, steps (or operators), step ordering constraints, and causal links. This representation, referred to as the plan structure, is updated throughout the planning process. When all the goals in plan structure are supported by non-conflicting steps then the structure represents a partially ordered plan to achieve the specified goals.

On each iteration, the planner selects one "open" goal and computes all possible ways of achieving it. There are three ways to satisfy a goal: add a new step to the plan, link the goal to an effect of an existing step, or link the goal directly to the current state. One of these options is added the plan structure and a choice point created with the alternatives. If we ever backtrack to this choice point then the previously chosen option is removed and one of the untried alternatives is inserted in its place. When a step is added to the plan all of its preconditions become "open" goals which must be "closed" in future iterations. Adding a step does not involve specifying when the step is to be executed. Partial step orderings are only added to the plan to resolve a "threatened link", or possible goal clobbering. In general, whenever the effect of any step in the plan threatens to clobber any closed goal, a new choice point is established with all possible ways of resolving the threat. If the plan structure contains an unresolvable threat, or there are no options to achieve the selected goal, the current plan structure fails and we backtrack to the last choice point.

We have weakened SNLP in several ways. First, we do not allow the possibility of closing a goal by link-

[3]Imagine logging in to a remote machine, whose directory hierarchy is unknown to you, and trying to map out the appropriate series of `cd` and `ls` commands.

ing to an existing step. Second, we have hardwired in depth-first search while true SNLP can accommodate various search strategies. We get depth-first search by using chronological backtracking. We continue to extend a plan until it fails (for any reason) and then backtrack to the most *recent* choice point with untried alternatives. We then try one of those alternatives and begin to move forward again.

Our main extension to SNLP is that, at the beginning of every iteration, one or more of the steps in the current plan may be executed. The current plan structure is passed through a special function that determines which steps to execute. If no steps are executed then our algorithm is equivalent to the SNLP algorithm. There are several possible results of an execution. One is that the execution of the step may fail. This suggests a defect in the operator model or in the softbot's model of the world. We expect that the softbot will eventually be able to aggressively investigate these possibilities. At the moment, however, an execution failure simply triggers chronological backtracking.

If the execution is successful then the state is updated with the effects of the operator, the step is removed from the plan, and any goals the step was supporting are linked directly to the state. Also, the execution of an operator might reveal new bindings for variables in the plan. This usually happens, for example, when the softbot executes the UNIX `ls` command and learns about new files in some directory. In this case, a new choice point is established to reflect the information gained from execution. However, even a successful execution might fail to achieve the goal it was intended to satisfy. For example, given the goal of being in the root directory, the softbot might employ a sensory action to check if it already happens to be there. If, in fact, the softbot is elsewhere, then executing this sensory action does not support the goal, in which case the planner backtracks. Pseudo code describing our planning and execution algorithms more precisely appears in Figure 3.

SNLP is both sound and complete. However, adding an execution point plays havoc with these formal properties. SNLP is a partial-order planner but execution of a step imposes ordering constraints. Additionally, the discovery of new objects and modifications to the state during planning also open up new areas of potential incompleteness.

The Goal Manager

Most existing AI planners are unable to handle ongoing tasks or respond to exogenous events. Instead of trying to build a planner with these features, we have chosen to build a layer on top of our UWL planner that adds these features. This layer, the goal manager, interprets goals written in the Rodney Action Language (RAL) and uses the planner to achieve those goals.

RAL was designed around the notion of an action. An action is simply a task that the softbot knows how

The planner algorithm:

```
if choose to execute step S in PLAN
  then execute S
       update PLAN based on execution
       if execution of S failed
         then backtrack
else
  pick open goal G in PLAN
  create a choice point with options for closing G
  if there are threats in PLAN
    then create a choice point with options
         for resolving each threat
  if there are no open goals in PLAN
            or no options to close G
            or an unresolvable threat in PLAN
    then backtrack

Termination:
    Planning fails if backtracking fails
    Planning succeeds if goals are achieved in world
                      and all open conditions are closed.


 Executing a step S:
Let G = the condition that step S supports
execute S
record execution of S in backtrack stack of PLAN
remove S from plan
link G to state
if execution revealed new bindings for G
  then create a choice point with these bindings
return :execution-match-failure  if G not supported
       :execution-failure        if execution failed
       :success                  otherwise
```

Figure 3: Pseudo-code description of Rodney's planning and execution algorithm.

```
(select (?file)
    ((name ?file "to-print") . t))
(print-file ?file)

(select (?file)
    ((file.type ?file lisp) . t)
    ((string.in.file ?file "*dc*")))
(replace-string ?file "*dc*" "*plan*")
```

Figure 4: Some simple actions. The select statements provide arguments for actions.

to perform. An action can be executing an UWL operator, achieving a set of UWL goals by invoking the planner, or executing a procedure consisting of other actions. Thus, an RAL action can be viewed as a partially-specified plan. Some of the "subactions" are specified and other are left at the planner's discretion. RAL actions are specified in two parts. First, the objects that will serve as arguments to the action are determined. Second, the action to be performed on the objects is specified. This is illustrated by the simple actions appearing in Figure 4.

RAL currently supports a simple notion of universal quantification. Any action can result in binding a variable to a set of objects. Similarly, actions can be defined that take sets of objects as arguments. As an example, the Unix `ls` command returns the set of objects in the current directory. When an action is given a set, the goal manager will execute the action multiple times in order to achieve the desired effect. In particular, when the goal manager is given an UWL goal with a variable bound to a set, the goal manager will send the planner the same goal for each value of the variable. This effectively the same as giving the planner a universally quantified goal.[4]

One of the most important features of the goal manager is its ability to handle goals involving exogenous events. Common Unix goals such as waiting for a print job to complete, waiting for a user to log on, and monitoring bboards are easily expressed in terms of actions that are triggered by exogenous events. In RAL you can request that an action occur whenever a set of events occurs in the world. For instance you can specify that the **send-mail** action occur when the literal `((active.on neal june) . t)` becomes true. This example and others are illustrated in Figure 5.

The manner in which the goal manager handles exogenous events is a good example of its interaction with the planner. When the goal manager receives a request involving an exogenous event, it places the desired event on a list of events it must monitor. Periodically, it cycles through this list and one by one asks the planner to find out the status of each event's predicates. By using the planner, the goal manager can utilize all the planner's reasoning ability to determine whether an event has occurred[5] When an event is detected, the goal manager proceeds to execute the actions associated with that event, again using the planner as needed.

[4]This approach is quite similar to the approach taken in planners, such as PRODIGY [Minton *et al.*, 1989] and UCPOP [Penberthy and Weld, 1992], that handle universally-quantified goals.

```
(select (?f)
    ((name ?f "long-paper.dvi") . t))
(request (print-file ?long-paper)
 :when ((printer.status ?p idle) . t))

(select (?neal)
    ((preferred.name ?neal "neal") . t))
(select (?self)
    ((current.user softbot ?self) . t))
(select (?machine)
    ((machine.name ?machine "june") . t))
(request (send-mail ?self "Neal logged in.")
 :when ((active.on ?neal ?june) . t))
```

Figure 5: Sample actions involving asynchronous events.

```
(request (display "Printer out of paper.")
 :when ((printer.status ?p no-paper) . t)
 :duration ((job.status ?job completed) . t))

(request (monitor-bboards ?bboards ?topics)
 :when ((new-messages ?bboards ?topics) . t)
 :duration continuous)
```

Figure 6: Sample continuous actions with durations.

Many tasks presented to Rodney involve continuous behavior. For instance monitoring bboards requires continuously checking them for new messages. Some goals requiring continuous behavior may only last for a specific period of time. A goal such as monitoring printer errors may only require continuous behavior while the user has an active print request. In RAL you can specify the duration of an action as once, continuous, or continuous until a particular event occurs. The duration of an action defaults to once, this prevents the actions in Figure 5 from occurring multiple times. Examples of actions that continuously respond to events appear in Figure 6.

In figure 7 we see examples of the final class of goal RAL can express. Maintain goals indicate conditions that Rodney should constantly try to keep true in the world. The first goal in the example expresses the goal of insuring that all files within a directory are group writable; the second expresses the goal that all lisp

[5]An event occurs when the corresponding literal changes its truth value.

```
(ls ?dir ?files)
(maintain
    (satisfy ((group.readable ?files) . t))
    (satisfy ((group.writable ?files) . t)))

(ls ?dir ?files)
(maintain
    (find-out ((type ?files lisp) . t))
     (satisfy ((dec.compiled ?files) . t))
     (satisfy ((sparc.compiled ?files) . t)))
```

Figure 7: . Sample actions with maintain goals.

files in the directory are compiled for each cpu type. Although not shown in the examples, it also possible to specify a duration for maintain goals as done with any ongoing task.

The full power of RAL can be seen in the advanced print action that appears in Figure 8. The print operator uses the UWL planner to find out who is the current user and what printer he or she prefers. It then sends the file to that printer by executing the Unix `lpr` command. Finally, it spawns off several background tasks to monitor the printer. Anytime an error occurs or the printer runs out of paper while the print job is still going, the user will be notified. When the print job completes, the user is notified and Rodney stops monitoring the printer. This same print operator can be used to print multiple files by passing it a set of files as an argument. All of this adds up to a high level of functionality that would be very difficult to build directly into a planner.

Future Work

To make Rodney easier to access, we are developing an e-mail interface to the softbot. In addition, we are exploring a number of learning problems including:

- Learning UWL models of UNIX commands from experiments.
- Locating, and automatically learning to access, Internet information resources.
- Automatic customization to individual users via high-level dialog.

In addition, we plan to extend Rodney's capabilities to access the Internet via the world-wide web, and to perform useful tasks such as filtering e-mail, providing an Internet white-pages service, and more.

Related Work

Work related to our UNIX softbots project falls into two broad categories: work on software agents, outside AI, that does not attempt to endow the agents with AI capabilities such planning and learning, and work inside AI that focuses on various aspects of the softbot

```
(defaction print-file (?file)
  (select (?user)
      ((current.user softbot ?user) . t))
  (select (?printer)
      ((preferred.printer ?user ?printer) . t))
  (lpr ?file ?printer)
  (select (?filename)
      ((name ?file ?filename) . t))
  (select (?job)
      ((job.status ?job working) . t)
      ((job.printer ?job ?printer) . t)
      ((job.user ?job ?user) . t)
      ((job.name ?job ?filename) . t))
  (request (display "Printer out of paper !!!")
   :when ((printer.status ?printer no-paper) . t)
   :duration ((job.status ?job completed) . t))
  (request (display "Printer error !!!")
   :when ((printer.status ?printer error) . t)
   :duration ((job.status ?job completed) . t))
  (request (display "Print job completed !!!")
   :when ((job.status ?job completed) . t)))
```

Figure 8: Sample print action that notifies the user when errors occurs.

problem. Due to space constraints, we provide a very brief discussion of some related AI projects.

Dent *et al.* [Dent *et al.*, 1992] describe CAP "a learning apprentice for calendar management." CAP provides a convenient appointment management tool, that facilitates scheduling appointments. CAP's actions manipulate a calendar by adding, deleting, and moving appointments. CAP's environment consists of the calendar it maintains and the commands it receives from the human user. Its effectors are commands that update the calendar; CAP senses the calendar's state to determine which operations are allowable at any given point. CAP does not have explicit goals or planning capabilities, but it does learn to suggest default values for various choices (e.g., meeting duration and location). Future plans for CAP include allowing it to arrange, confirm, and re-schedule meetings.

Maes *et al.* [Maes and Kozierok, 1993] are developing a number of *interface agents*—application-specific user interfaces with learning capabilities. There are two central differences between this work and our own. First, we have focused on providing Rodney with general-purpose planning, execution, and representation facilities that enable to interact with the wide variety of applications, commands, and programs found in the UNIX environment. Second, Maes *et al.* are focusing on a knowledge-lean approach to learning where the interface agent requires as little information as possible about the task domain, and learns in an unsupervised manner. We plan to focus on knowledge-intensive learning that is done in cooperation with the user (or with a domain expert).

Shoham's work on the agent-oriented programming (AOP) framework [Shoham, 1993] is complementary to our own and to much of the work described above. Whereas we have taken an empirical approach—developing agents that tackle useful tasks in the UNIX domain—Shoham is explicating terms such as "agent," "commitment," "choice," and "capability" as basis for designing (both software and hardware) agents and communication protocols between them.

The UNIX Consultant (UC) is a natural-language interface that answers naive user queries about UNIX. The UC project focused on identifying the user's plans, goals, and knowledge. UC utilized this information to generate informative responses to queries. For example, if the user asked "is rn used to rename files?" UC not only told her "No, rn is used to read news," but also said that the appropriate command for renaming files is "mv." Based on the query, UC hypothesized that the user's goal is to rename a file and decided that the information regarding "mv" is relevant. In contrast to Rodney, the UC does not act to achieve its own goals, but responds to user's queries.[6] Although the UC had a limited capability to learn about UNIX commands from natural-language descriptions, it did not have the capacity to explore its environment or actually execute any UNIX commands. In short, the UC was a sophisticated natural-language interface, not a softbot. The UC project demonstrated the fertility of UNIX as a real yet manageable domain for AI research, a lesson we have taken to heart.

Conclusion

Software environments (e.g. distributed databases, computer networks) are gaining prominence outside AI, demonstrating their intrinsic interest. Building agents that perform useful tasks in software environments is easier than building the corresponding agents for physical environments. Thus, softbots are an attractive substrate for AI research, resolving the potential conflict between the drive for integrated agents operating in real-world task environments and the desire to maintain reasonable progress in AI. At the same time, incorporating AI capabilities into software tools has the potential to yield a new class of software technology with planning and learning capabilities.

To support these claims we have described our ongoing project to develop UNIX softbots. The project is still in its infancy, but our softbots are already producing both interesting behavior and thought-provoking research challenges.

References

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

[Dent *et al.*, 1992] Dent, Lisa; Boticario, Jesus; McDermott, John; Mitchell, Tom; and Zabowski, David 1992. A personal learning apprentice. In *Proc. 10th Nat. Conf. on Artificial Intelligence.* 96–103.

[Etzioni and Weld, 1994] Etzioni, O. and Weld, D. 1994. The first law of robotics. Technical report, Univ. of Washington, Dept. of Computer Science and Engineering. Forthcoming.

[Etzioni *et al.*, 1992] Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning.* Available via anonymous FTP from `~ftp/pub/ai/` at `cs.washington.edu`.

[Etzioni *et al.*, 1993] Etzioni, Oren; Lesh, Neal; and Segal, Richard 1993. Building softbots for UNIX (preliminary report). Technical Report 93-09-01, University of Washington. Available via anonymous FTP from `~ftp/pub/ai/` at `cs.washington.edu`.

[Etzioni *et al.*, 1994] Etzioni, O.; Golden, K.; and Weld, D. 1994. Tractable closed-world reasoning with updates. Technical Report 94-01-02, University of Washington, Department of Computer Science and Engineering.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4).

[Golden *et al.*, 1994] Golden, K.; Etzioni, O.; and Weld, D. 1994. XII: Planning for Universal Quantification and Incomplete Information. Technical report, University of Washington, Department of Computer Science and Engineering.

[Maes and Kozierok, 1993] Maes, Pattie and Kozierok, Robyn 1993. Learning interface agents. In *Proceedings of INTERCHI-93.*

[McAllester and Rosenblitt, 1991] McAllester, D. and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. on Artificial Intelligence.* 634–639. internet file at ftp.ai.mit.edu: /pub/users/dam/aaai91c.ps.

[Minton *et al.*, 1989] Minton, Steven; Carbonell, Jaime G.; Knoblock, Craig A.; Kuokka, Daniel R.; Etzioni, Oren; and Gil, Yolanda 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40:63–118. Available as technical report CMU-CS-89-103.

[Olawsky and Gini, 1990] Olawsky, D. and Gini, M. 1990. Deferred planning and sensor use. In *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control.* Morgan Kaufmann.

[Penberthy and Weld, 1992] Penberthy, J.S. and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning.* 103–114. Available via anonymous FTP from `~ftp/pub/ai/` at `cs.washington.edu`.

[Shoham, 1993] Shoham, Y. 1993. Agent-oriented programming. *Artificial Intelligence* 60(1):51–92.

[6] The UC did note when the user's goals were in conflict with its internal agenda and refused to answer queries such as "how do I crash the system?"