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

November 1992

"If you want to think about thinking, you have to think about thinking about something." — Seymour Papert

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 systematically 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*)—intelligent agents that interact with UNIX. Existing softbots 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.

*This is an abridged version of a report we began to circulate in November 1992, describing our first fully-implemented softbots. Since that time, we have incorporated a more powerful planner into the softbot [Etzioni *et al.*, 1993a, Golden *et al.*, 1993], 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 2), and the spirit of our investigation, remain the same.

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

1 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 systematically experiment with software artifacts are relatively low. Software also facilitates the dissemination and replication of research results. Finally, developing effective sensors and actuators is relatively easy, making software environments particularly attractive for agent research.
  - *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:* though rich, 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.

  A more detailed argument for this claim, aimed at a general computer-science audience, appears in [Etzioni *et al.*, 1993b].

The above claims are empirical. We set out to refine and validate the claims by developing *softbots* (*soft*ware ro*bots*) for UNIX [Etzioni and Segal, 1992]. Our twin goals are to make fundamental contributions to core AI, relying on UNIX as a testbed, and to develop technology that will actually assist UNIX users.

The remainder of this report is organized as follows. Section 2 describes the notion of a softbot in more detail. The following section presents our very first softbot—St. Bernard (a softbot that specializes in locating and retrieving "lost" files). The bulk of the report focuses on Rodney, a more sophisticated, general-purpose UNIX softbot that we are continuing to extend and improve in many ways. We conclude with a discussion of related and future work.

2 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 meant to change the external environment's state (e.g. commands such as `mv` or `compress` in UNIX). A softbot's sensors are commands meant to provide the softbot with information about the environment (e.g. `pwd` or `ls` in UNIX). 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 elements of "softbotics."

Some will argue that programs such as computer viruses or even shell scripts are degenerate cases of softbots. However, the fundamental difference between softbots and other programs is the commitment to AI capabilities inherent in the softbot paradigm. We envision softbots that possess the following capabilities:

1. **Goal-directed behavior** —a softbot attempts to achieve explicit goals. Unlike a "Brooksian Creature," it does not merely following 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. For example, our UNIX softbot represents "retry" as a general error recovery strategy. Thus, whenever an action fails, the softbot has the option of retrying the action at some later point in time. The softbot treats the questions "should a particular action be retried and, if so, how often?" as learning problems to be resolved by experience.

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. location of the Lisp executable) and the preferences of its human partners.

5. **Continuous operation** — A softbot continuously operates 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. This ability suggests a variety of strategies for survival, exploration, and learning. We are only beginning to understand the relationship between cloning and intelligence.[2]

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

3 St. Bernard: a File-Retrieving Softbot

St. Bernard is a softbot that is able to locate and retrieve files based on a partial description of their characteristics [Segal, 1992]. When the description includes the file's parent directory, this task is straight forward but, in many cases, the file's parent directory and name are unknown. Users often find themselves asking questions such as "where did I put my MLC '88 paper?" or "where is Allegro Common Lisp on this machine?" Queries of this sort can lead to a massive search for the appropriate file.

Humans handle such searches by relying on expectations regarding the location of the target file ("it's probably somewhere under my `/papers` directory"). St. Bernard is a file-retrieving softbot that acquires expectations regarding file location from experience, and attempts to minimize its average search cost by relying on these expectations. St. Bernard expresses its expectations as probabilities: how likely is the target file to be in a given directory?

St. Bernard sorts the directories in the hierarchy and searches them in sequence, stopping when the target file is found. All other things being equal, St. Bernard prefers to search directories more likely to contain the target file. This preference must be tempered by estimates of directory size. To illustrate this point, consider searching for the file f in the directories d_1 and d_2; Suppose $P(f \in d_1) = 0.8$ and $P(f \in d_2) = 0.9$, but directory d_1 contains only five files and d_2 contains one thousand files. Since f is more likely to be in directory d_2, St. Bernard might be tempted to search d_2 followed by d_1. The expected cost of this strategy is:

$$0.9 * 1000 + (1 - 0.9) * 5 = 900.5.$$

The expected cost of the alternative strategy, d_1 followed by d_2, is much smaller:

$$0.8 * 5 + (1 - 0.8) * 1000 = 204.$$

The small cost of searching d_1 outweighs the reduced chance of finding f there. Thus, St. Bernard has to balance the probability of finding a file in a given directory with the cost of searching that directory.

To explain St. Bernard's strategy more precisely, we need to introduce some notation:

D	—	The set of directories to search.
f	—	The description of the file to be found.
$P(f \in d)$	—	The probability that searching the directory d will succeed.[3]
$C(d)$	—	The cost of searching the directory d.
σ	—	A sequence $(d_1, d_2, d_3, ..., d_n)$ to search the directories in D.

[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.

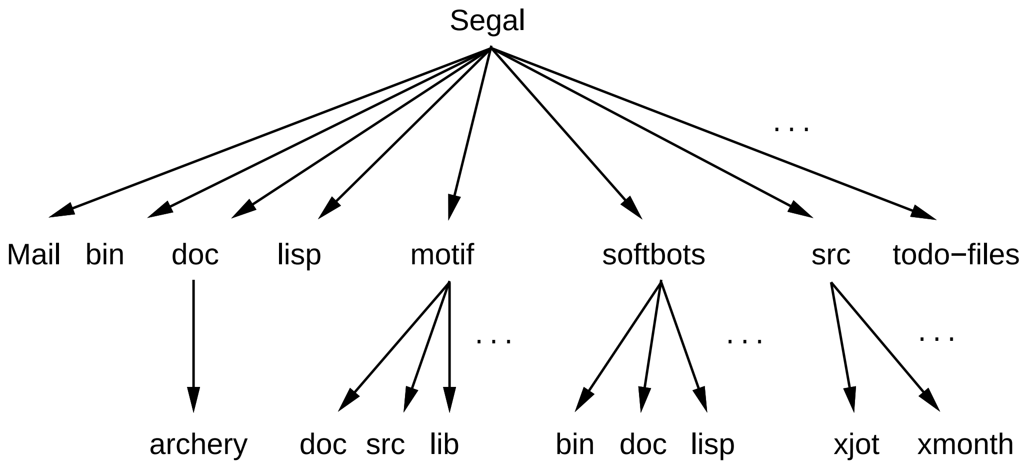

Figure 1: A sample directory hierarchy searched by St. Bernard.

If we assume that the probability of finding a file matching the description f, in a given directory d_i, is independent of the probability of finding such a file in other directories, then the expected cost of St. Bernard's search is:

$$EC(\sigma) = \sum_{d_i \in \sigma} C(d_i) \prod_{j=1}^{i-1} (1 - P(f \in d_j))$$

This formula weights the cost of searching each directory by the probability that the directory will be searched. A directory is searched only if St. Bernard failed to find the target file f in a directory searched earlier in the search sequence. An optimal search ordering σ^* is an ordering of D that minimizes the expected search cost. Naively, we could determine σ^* by computing the expected cost of all directory orderings, but for k directories this takes $O(k!)$ time. As proved in [Etzioni, 1991, Simon and Kadane, 1975] and elsewhere, we can compute σ^* in $O(k \log k)$ time by sorting the directories of D in decreasing order on $\frac{P(f \in d)}{C(d)}$.

To apply this procedure in practice, St. Bernard has to estimate the probabilities and costs involved. St. Bernard can estimate $C(d)$ based on previous experience with the directory, or based on the number of kilobytes in the directory. To compute $P(f \in d)$, St. Bernard maps f to a file class such as "TEX files owned by Segal" or "files created before March 1, 1991" and estimates the proportion of files in d that belong to the class based on its previous searches in the directory. St. Bernard supports arbitrary classification schemes by making every attribute of a file available for classification purposes. See [Segal, 1992] for a more comprehensive description of St. Bernard.

Given the directory hierarchy shown in Figure 1, St. Bernard searched for the file `xjot.c` in the directories `/src, /motif, /softbots` in order. St. Bernard omitted directories such as `/Mail` since it believes the probability of finding a C-file in such directories is zero. When searching for the file `softbots.tex`, St. Bernard searched `/doc /softbots` in order.

We tested St. Bernard on the directory hierarchy shown in Figure 1. After exploring the directory hierarchy, to form its probability and cost estimates, St. Bernard is given a random

[3]Clearly, the file f is either in d or not, but P can be interpreted as St. Bernard's degree of belief in the proposition $f \in d$.

sample of target files. Table 1 summarizes St. Bernard's performance and compares it with an exhaustive search of the hierarchy in which the directories are searched in alphabetical order.

Class	Search in alphabetical order	St. Bernard	Speedup
$\TeX$	50.1 (5.0)	25.0 (1.7)	2.0
LISP	97.8 (8.8)	28.8 (1.4)	3.4
C	527.1 (7.3)	177.5 (1.4)	3.0
Totals	675.0 (7.2)	231.3 (1.6)	2.9

Table 1: Total times, in CPU seconds, for locating a random set of target files in Segal's directory hierarchy. Parenthesis indicate the average number of directories visited. Since visiting a single directory is optimal, we see that St. Bernard's searches are quite close to optimal.

3.1 Discussion

In essence, we have formulated the file-retrieval problem in the classical framework of *satisficing search* (i.e., minimizing expected search cost) introduced by [Simon and Kadane, 1975]. Simon and Kadane's analysis was purely theoretical. Algorithms for deriving cost and probability estimates from experience were proposed and analyzed theoretically in [Etzioni, 1991], and simplified versions were implemented as part of the SE system [Mitchell *et al.*, 1991], but St. Bernard represents the first attempt we are aware of to empirically test satisficing search in a real-world domain. The abundance of data for "training" St. Bernard, and the presence of a real-world yet easily-studied task demonstrates the pragmatic convenience of softbots as experimental testbeds for AI.

In future work we plan to integrate St. Bernard's estimates of probability and costs with symbolic rules (e.g., search for system files in system directories, but not in user directories) provided by the user or learned from experience. This approach will enable us to empirically test and extend the recent work on learning algorithms that combining theories and data (e.g., [Ourston and Mooney, 1990, Pazzani *et al.*, 1991]). We also plan to generalize St. Bernard, so it is able to move beyond file-retrieval tasks, and minimize the expected cost of searches for available printers, free machines, users who need to be contacted, etc.

4 Rodney: A General-Purpose UNIX Softbot

This section presents Rodney, a general-purpose UNIX softbot. We can decompose the low-level tasks Rodney is able to accomplish into three broad categories:

- **Monitoring events:** immediately display on my screen any mail message I receive that contains the word "urgent."
- **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 mega-bytes and have not been modified in more than a week.

These task classes are neither exhaustive nor mutually exclusive, but illustrate our main point: the softbot enables a user to specify *what* to accomplish, in a high-level goal language, 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 user interfaces are incapable of such expressiveness.

The power provided by the softbot in each of the individual examples above could be achieve by writing a UNIX shell script. However, to match the full power of the softbot 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 softbot knows about a new facility, that facility becomes immediately available to its planning process, and is automatically invoked to satisfy relevant user requests.

Below, we describe Rodney's architecture and provide examples of Rodney "in action." Rodney's architecture is shown in Figure 2. The architecture has four major components:

- **Goal manager:** receives task specifications from the user, and periodically invokes the planner with goals to achieve.
- **Planner:** satisfies goals by interleaving planning and execution. The planner is discussed in more detail in Section 4.2, and its algorithm appears in Table 3.
- **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.
- **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.

4.1 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 [Hintikka, 1962, Moore, 1985, Morgenstern, 1988] have not yielded planning algorithms or implemented planners.

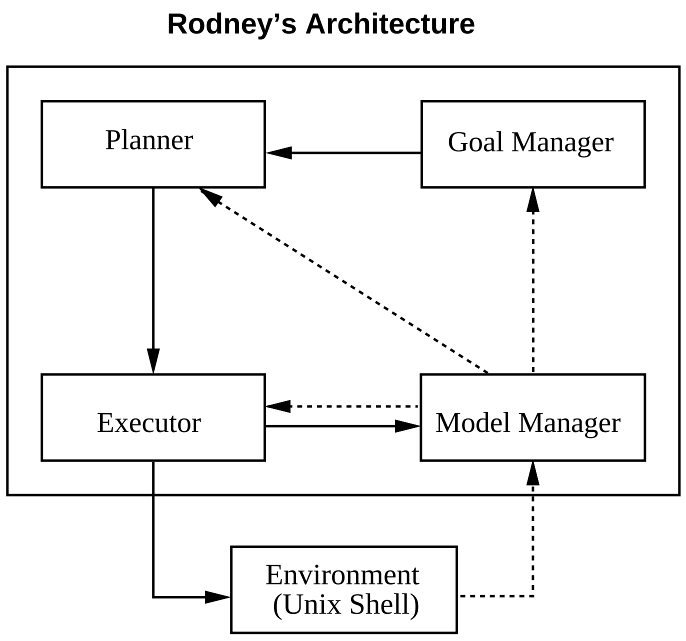

Figure 2: A high-level view of Rodney's Architecture. Loosely speaking, the solid arrows represent control flow in Rodney, and the dashed arrows indicate data flow.

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 Table 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))
```

Table 2: UWL representation of the UNIX command `wc` which provides the character, word, and line count of a file.

4.2 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 [Etzioni *et al.*, 1992, Schoppers, 1987], 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.[4] Making assumptions is appropriate in some cases (e.g., we may assume that a file is appropriately protected, and replan if we get an

[4]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.

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, Barrett and Weld, 1993]. 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 linking 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 Table 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. See [Lesh, 1992, Etzioni and Lesh, 1993] for further discussion of the planner's design.

```
until a termination condition is reached:
    if choose to execute step S in PLAN
      then execute S
           if execution of S succeeded
             then update PLAN based on execution
           if execution of S failed
             then backtrack
    else
      pick open condition 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 were no open conditions 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 conditions are achieved in world
```

Table 3: Pseudo-code description of Rodney's planning algorithm.

4.3 Rodney in action

To make the planner's operation more concrete, this section shows how Rodney satisfies a simple conjunctive goal:

```
(find-out ((pathname ?dir ''/ai/softbots/test/bin'') .  T))
(find-out ((parent.directory ?file ?dir) .  T))
(find-out ((group.protection ?file readable) .  F))
```

This goal asks Rodney whether there is a file in the `/ai/softbots/test/bin` directory that is not group-readable. Below, we explain Rodney's behavior in detail.

By default, Rodney closes goals in the order the goals are presented. So first it chooses the `pathname` subgoal. From previous experience, Rodney is familiar with the `/ai/softbots/test/bin`, so it binds the variable `?dir` to the object representing that directory. Rodney's very first choice illustrates the benefits of UWL. The `mv` operator could potentially satisfy the `pathname` subgoal, by renaming a directory to be `/ai/softbots/test/bin`, but Rodney rules out `mv` due to the subgoal's `find-out` annotation. Rodney wants to access the directory named `/ai/softbots/test/bin`, not a *different* directory whose name has been changed to `/ai/softbots/test/bin`. Rodney could use the `local-ls` operator to satisfy this goal. However, a control rule rejects this possibility when there is a binding choice available.

Next, Rodney decides to satisfy `(find-out ((parent.directory ?file ?dir) . T))` using the operator `local-ls`, which is the only operator that can potentially satisfy this goal. The precondition of of the `local-ls` operator , `(satisfy (current.directory rodney ?dir) . T)`, becomes an open goal in the plan. Rodney chooses this precondition. Since Rodney does not know its current directory it has two alternatives: `pwd` and `cd`. In general, a `satisfy` goal can be achieved either by an operator with an `observe` postcondition, such as `pwd`, or an operator with a `cause` postcondition such as `cd`. Rodney creates a choice point with these two alternatives. First it chooses to add the `pwd` operator to the plan. This operator can be executed because all of it's preconditions are linked directly to the state. Rodney executes `pwd` and finds that its `current.directory` goal is not satisfied. This failure illustrates a distinctive feature of sensory actions. Unlike classical STRIPS operators: A sensory action can fail to satisfy its goal even when its preconditions are satisfied and it is successfully executed.

Since `pwd` failed, Rodney backtracks to its last choice point and considers `cd` next. Rodney executes `cd`. Now the preconditions of `local-ls` are also linked to the state and so it executes `local-ls` and discovers 3 files in the directory `/ai/softbots/test/bin`. Rodney now creates a choice point with these three bindings for the `?file` variable. At this point, Rodney tackles the final top-level subgoal `(find-out ((group.protection ?file readable) . F))`, and chooses the operator `protection-on-file` ("ls -l" to UNIX affecionados) to satisfy this goal.

The first file that Rodney executes `protection-on-file` turns out to be group-readable. Thus Rodney backtracks to the choice point described above and tries the second binding for `?file`. This file is not group-readable so planning ends and Rodney returns success.

4.4 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, accepts a wide range of tasks, written in the Rodney Action Language (RAL), decomposes these into a set of actions to be executed and a set of UWL goals, and periodically invokes the planner to satisfy these goals.

RAL is designed around the notion of an *action*. There are two kinds of actions (both treated uniformly by the goal manager):

- **Primitive actions:** the main type of primitive action is the execution of a UWL operator. The goal manager ascertains that the operator's preconditions are satisfied by invoking the planner with the preconditions as a top-level goal. Once the preconditions are satisfied, the goal manager instructs the executor to issue to execute the operator. Additional primitive actions include asserting a fact, selecting an object, executing a lisp function, and more.

- **Compound actions:** a compound action is some combination of primitive actions. Legal combinations include sequences, conditionals, and loops.

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

(select (?file) ((file.type ?file lisp) . T)
                ((string.in.file ?file "*dc-print*") . T))
(replace-string ?file "*dc-print" "*planner-print*")
```

Figure 3: Some simple actions. The select statements provide arguments for actions.

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

(select (?neal) ((preferred.name ?neal "neal") . T))
(select (?machine) ((machine.name ?machine "june") . T))
(request (double-beep)
        :when ((active.on ?neal ?june) . T))
```

Figure 4: Sample actions involving asynchronous events.

In general, an RAL action can be viewed as a partially-specified plan. Some of the primitive actions may be specified, but others 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 3.

RAL currently supports a simple notion of universal quantification. Any action can result in binding a variable to a set of objects. For 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 iterate through the set, repeatedly sending the planner instances of the goal. Each element in the set yields a distinct goal that is sent to the planner.[5]

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 `double-beep` action occur when the literal `((active.on neal june) .  T)` becomes true. See Figure 4 for examples.

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

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

```
(request (display "Printer out of paper.")
        :when ((printer.status ?printer out-of-paper) . T)
        :duration ((job.status ?job completed) . T))

(request (monitor-bboards ?bboards ?topics)
:when ((new-messages ?bboards ?topics) . T)
:duration always)
```

Figure 5: Sample of repetitive actions.

```
(assert ((pathname ?dir ''/softbots/rodney/stable'') . T))
(ls ?dir ?files)
(maintain (satisfy ((group.protection ?files readable) . T))
          (satisfy ((group.protection ?files writable) . T)))

(maintain (find-out ((file.type ?files lisp) . T))
          (satisfy ((compiled ?files decstation) . T))
          (satisfy ((compiled ?files Sparcstation) . T)))
```

Figure 6: . Sample actions with constraint goals.

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[6] When an event is detected, the goal manager proceeds to execute the actions associated with that event, again using the planner as needed.

Many tasks presented to Rodney involve looping behavior. For instance, monitoring bulletin boards requires repeatedly checking for new messages. Some tasks that require looping have limited duration. For instance, a goal to monitor whether a job has completed is only of interest while that job is active. In RAL you can specify the duration of an action as `once`, `always`, or "repeat until a particular event occurs." The duration of an action defaults to `once`, which prevents the actions in Figure 4 from occurring multiple times. Examples of repeated actions appear in Figure 5.

In figure 6 we see examples of the final class of goal RAL can express: *constraint goals* indicate conditions that Rodney should constantly try to maintain in the world. The first goal in the example expresses the goal of ensuring that all files within a directory are both group-writable and group-readable; the second expresses the goal that all lisp files in the directory are compiled for two machine types. Although not shown in the examples, it also possible to specify a duration for constraint goals as done with any ongoing task.

Due to the dynamic nature of the UNIX environment, information frequently becomes out of date. This problem is particularly acute for constraint maintenance and event monitoring tasks, where the point of the task is to repeatedly *check* whether a certain condition is maintained or

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

```
(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 out-of-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 7: Sample print action that notifies the user when errors occurs.

whether a certain event has occurred. If the condition is only checked against the softbot's world model, then the softbot will overlook both exogenous events and unforeseen effects of its own actions. To address this problem we introduce the `sense` goal annotation. The goal

```
(sense ((parent.dir ?file /joint-paper) .  T))
```

is an information goal that can only be satisfied by executing a sensory action. Thus, when a `sense` goal is satisfied, the softbot is guaranteed that its information is current. The goal manager maps the "event triggers" in the `:when` and `:duration` fields of RAL actions to `sense` goals, and periodically invokes the planner to satisfy these goals.

In conclusion, we illustrate the power of RAL via the sophisticated print action shown in Figure 7. The print action 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 (invoking the planner to satisfy `lpr`'s preconditions, if necessary). 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. Thus, RAL enables us to naturally encode high-level behavioral specifications that would be difficult to build directly into a planner.

The design of Rodney forced us to confront the problem of planning with incomplete information in a dynamically changing environment. The core of our solution to this problem are the UWL representation [Etzioni *et al.*, 1992] which facilitates planning with information goals and sensory actions, and the RAL representation, which facilitates responding to exogenous events, maintaining constraints, and receiving partially-specified plans from the user.

5 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 problem. By and large, work outside AI has focused on *knowbots* [Kahn and Cerf, 1988], agents that mediate between humans and vast knowledge stores.[7] In essence, a knowbot is a sophisticated interface to a network of distributed databases. One knowbot can accept a query, translate it into a variety of formats, and transmit the translated queries to various knowbots on the network, which are then expected to reply. Important concerns in this context are billing and inter-agent communication protocols. Prototype knowbots are hard-coded to perform certain functions (e.g., look up user addresses [Droms, 1990]). In contrast to softbots, existing knowbots do not plan or learn from their experience.[8] In a sense, softbots can be viewed as general-purpose knowbots with AI capabilities.

5.1 AI Work on Software Agents

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).

Steier and Newell [Newell and Steier, 1991] describe preliminary work on interfacing Soar to a variety of software packages such as Mathematica, a drawing package, and more. The Soar project has focused on how to structure the interaction within Soar's architectural constraints (e.g., universal subgoaling, automatic chunking, etc.). For instance, Doorenbos *et al.* [Doorenbos *et al.*, 1992] describe the performance of a Soar system that learns more than ten-thousand chunks by interacting with a database. Like the CAP project, the emphasis in this work has been on learning rather than planning.

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 particularly

[7]Knowbots are a trademark of the corporation for national research initiatives.

[8]Planning and symbolic learning capabilities distinguish softbots from various "artificial life" programs [Langston *et al.*, 1989] as well.

concerned with developing logics that provide precise definitions for terms such as "agent," "commitment," "choice," and "capability" as basis for designing (both software and hardware) agents and communication protocols between them.

Over the years, the UNIX domain has periodically attracted attention in AI. We briefly contrast our UNIX softbots with two major projects: Wilensky *et al.*'s UNIX Consultant [Wilensky *et al.*, 1988], and Dietterich's EG system [Dietterich, 1984].

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.[9] 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.

Dietterich's EG system [Dietterich, 1984] represents an ambitious attempt to use UNIX as a test domain for theory formation. EG focuses specifically on the problem of data interpretation which, in UNIX terms, involves correctly understanding the string returned by the UNIX shell in response to a UNIX softbot's action. In contrast, our softbots start with models of UNIX output that enable them to correctly parse it into high-level logical descriptions, in most cases. In essence, EG is a knowledge-lean system that attempts to acquire, using learning techniques, much of the knowledge that we "hand feed" our softbots.

6 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 useful behavior and thought-provoking research challenges.

Acknowledgments

We thank Denise Draper, Steve Hanks, Craig Knoblock, Nick Kushmerick, Steve Minton, Tom Mitchell, Dan Weld, Mike Williamson, and members of the UW AI group for helpful discussions.

[9] 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?"

Thanks are also due to Steve Hanks, Dan Weld, Denise Draper, and Mike Williamson for their collaboration in designing UWL. This research was funded in part by Office of Naval Research Grant 92-J-1946, and by National Science Foundation Grants IRI-9211045 and IRI-9357772.

References

[Barrett and Weld, 1993] Barrett, A. and Weld, D. 1993. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence.* To appear. Available via anonymous FTP from `~ftp/pub/ai` at `cs.washington.edu`.

[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 *Proceedings of AAAI-92*, Sam Mateo, California. Morgan Kaufmann. 96–103.

[Dietterich, 1984] Dietterich, Thomas Glen 1984. *Constraint Propagation Techniques for theory-driven data interpretation.* Ph.D. Dissertation, Stanford University.

[Doorenbos *et al.*, 1992] Doorenbos, R.; Tambe, M.; and Newell, A. 1992. Learning 10,000 chunks: what's it like out there. In *Proceedings of the National Conference on Artificial Intelligence.*

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

[Etzioni and Lesh, 1993] Etzioni, Oren and Lesh, Neal 1993. Planning with incomplete information in the unix domain. In *Working Notes of the AAAI Spring Symposium on Foundations of Automatic Planning*, Menlo Park, CA. AAAI Press.

[Etzioni and Segal, 1992] Etzioni, Oren and Segal, Richard 1992. Softbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA. AAAI Press. Also in Proceedings of the Canadian Machine Learning Workshop, Vancouver, B.C. 1992.

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

[Etzioni *et al.*, 1992] Etzioni, Oren; Hanks, Steve; Weld, Daniel; Draper, Denise; Lesh, Neal; and Williamson, Mike 1992. An Approach to Planning with Incomplete Information. In *Proceedings of KR-92.* Available via anonymous FTP from `~ftp/pub/ai/` at `cs.washington.edu`.

[Etzioni *et al.*, 1993a] Etzioni, O.; Golden, K.; and Weld, D. 1993a. Tractable closed-world reasoning with updates. Technical Report 93-xx-xx, University of Washington, Department of Computer Science and Engineering.

[Etzioni *et al.*, 1993b] Etzioni, Oren; Levy, Hank; Segal, Rich; and Thekkath, Chandramohan 1993b. OS agents: Using AI techniques in the operating system environment. Technical Report 93-04-04, University of Washington.

[Etzioni, 1991] Etzioni, Oren 1991. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence* 49(1–3):129–160.

[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.*, 1993] Golden, K.; Etzioni, O.; and Weld, D. 1993. XII: Planning for Universal Quantification and Incomplete Information. Technical report, University of Washington, Department of Computer Science and Engineering.

[Hintikka, 1962] Hintikka, Jaako 1962. *Semantics for Propositional Attitudes.* Cornell University Press, Ithica, N.Y.

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

[Langston *et al.*, 1989] Langston, C.; Farmer, D.; and Rasmussen, S., editors 1989. *Proceedings of an interdisciplinary workshop on the sysnthesis and simulation of living systems.* Addison-Wesley.

[Lesh, 1992] Lesh, Neal 1992. A planner for a UNIX softbot. Internal report.

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

[McAllester and Rosenblitt, 1991] McAllester, D. and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of AAAI-91.* 634–639. internet file 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.

[Mitchell *et al.*, 1991] Mitchell, Tom M.; Allen, John; Chalasani, Prasad; Cheng, John; Etzioni, Oren; Ringuette, Marc; and Schlimmer, Jeffrey C. 1991. Theo: A framework for self-improving systems. In VanLehn, K., editor 1991, *Architectures for Intelligence.* Lawrence Erlbaum, Hillsdale, NJ.

[Moore, 1985] Moore, R. 1985. A Formal Theory of Knowledge and Action. In Hobbs, J. and Moore, R., editors 1985, *Formal Theories of the Commonsense World.* Ablex, Norwood, NJ.

[Morgenstern, 1988] Morgenstern, Leora 1988. *Foundations of a Logic of Knowledge, Action, and Communication.* Ph.D. Dissertation, New York University.

[Newell and Steier, 1991] Newell, Allen and Steier, David 1991. Intelligent control of external software systems. Technical Report EDRC 05-55-91, Carnegie Mellon University.

[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.

[Ourston and Mooney, 1990] Ourston, Dirk and Mooney, Raymond J. 1990. Changing the rules: a comprehensive approach to theroy refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence.*

[Pazzani *et al.*, 1991] Pazzani, Michael; Brunk, C. A.; and Silverstein, G. 1991. A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Worskhop on Machine Learning.*

[Penberthy and Weld, 1992] Penberthy, J.S. and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92.* 103–114. Available via anonymous FTP from `~ftp/pub/ai/` at `cs.washington.edu`.

[Schoppers, 1987] Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *Proceedings of IJCAI-87.* 1039–1046.

[Segal, 1992] Segal, Richard 1992. St. bernard: The file retrieving softbot. Internal report.

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

[Simon and Kadane, 1975] Simon, Herbert A. and Kadane, Joseph B. 1975. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence* 6(3):235–247.

[Wilensky *et al.*, 1988] Wilensky, Robert; Chin, David; Luria, Marc; Martin, James; Mayfield, James; and Wu, Dekai 1988. The Berkeley UNIX Consultant project. *Computational Linguistics* 14(4):35–84.