Planning with Incomplete Information in the UNIX Domain*

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

1 Motivation

This paper serves two goals. First, we advocate real-world software environments, such as operating systems or databases, as domains for planning research. Second, we describe our ongoing research on planning with incomplete information. In both cases, we focus on the UNIX planning domain as a testbed for our implementation and a source of concrete examples and intuitions.

Real-world software environments, such as UNIX, have a number of attractive features as testbeds for planning research. First, it is relatively easy to develop "embodied agents," which interact directly with a software environment by issuing commands and interpreting the environment's feedback [6]. Such agents facilitate testing planners experimentally and integrating planning with execution and error recovery capabilities. Second, software environments circumvent many thorny, but peripheral, research issues that have to be addressed in physical environments (e.g., overcoming sensor noise, representing liquids, shapes, etc.). Third, software experiments are relatively easy to perform, control, and repeat, facilitating systematic experimental research of the sort advocated by [10] and others. Finally, software facilitates the dissemination and replication of research results. It is straightforward to distribute multiple copies of a software agent, and many software environments (e.g., UNIX) are easily accessible to most researchers.

Simulated worlds share some of these features but, software environments are readily available (sophisticated simulations can take years to develop and perfect). Furthermore, software environments are *real.* Finally, research in software domains has the potential to directly yield *useful* technology. The preponderance of problems such as *feature shock* (the paralysis a user feels when facing a bewildering array of complex, poorly-documented features [9]) and *information anxiety* (a user's response to the staggering volume and diversity of electronic data [21]) suggest that there is no shortage of useful tasks for intelligent software agents.

We refer to such agents as *softbots* (*soft*ware ro*bots*) [6].

- A softbot's *effectors* are commands transmitted to the external environment in order to change its state (e.g., UNIX shell commands such as `mv` or `compress`).
- A softbot's *sensors* are commands that provide it with information (e.g., `pwd` or `ls` in UNIX).

We have developed a softbot for a distributed UNIX environment [5, 6]. Below we describe some of the challenges the softbot's planner has confronted, and the solutions we have developed. Both readily generalize to other domains.

2 Incomplete Information

It is natural to think of certain UNIX shell commands such as `mv`, `cd` or `lpr` as operators, and of many 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

*We thank Richard Segal, our co-softboticist, for his numerous contributions to this research. Thanks are also due to Steve Hanks, Dan Weld, Denise Draper, and Mike Williamson for their collaboration in designing UWL, and for helpful discussions. This research was funded in part by Office of Naval Research Grant 92-J-1946, and by National Science Foundation Grant IRI-9211045.

to gather information. Following [14], we refer to such commands as *informative actions*. In order to satisfy its high-level goals, our UNIX softbot has to reason about what it does not know, and synthesize plans to obtain the necessary information using informative actions. Thus, the softbot faces the classical *knowledge preconditions* problem [13]. This problem motivated a number of highly-expressive logical axiomatizations of the notions of knowledge and belief [8, 14, 15], but the logicist approach has not yielded effective planning algorithms or implemented planners. Seeking to leverage classical planning techniques, we chose to devise the less expressive but more tractable UWL representation. In essence, UWL is the minimal extension to STRIPS, necessary to plan with incomplete information.

2.1 The UWL Representation

Classical planners (e.g., [3, 7]) presuppose correct and complete information about the world. Assuming correct information means that every proposition entailed by the planner's world model is in fact true in the world. The converse of this assumption is the assumption of complete information: every proposition that is true in the world is entailed by the planner's world model. In UWL we remove the completeness assumption, but continue to assume correctness.[1] The syntax and semantics of UWL are specified precisely in [5]. We provide a brief, intuitive discussion of the language below.

UWL divides an operator's effects into **cause** postconditions, which change the world's state, and **observe** postconditions that merely elaborate the agent's model of the world (cf. [4]). **cause** postconditions correspond to STRIPS adds and deletes. **observe** 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))`.[2] The variables `!v` and `!x` are *run-time* variables, which refer to particular values 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. Notions similar to run-time variables appear in [1, 16].

In addition, 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)` means roughly that the softbot ought to determine that `P` is true, without changing `P`'s state in the process. This constraint is critical to information gathering. Otherwise, if we give the softbot the goal `(find-out (name ?file core-dump))`, 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. Thus, the softbot typically supports **find-out** goals with **observe** postconditions. Informative UNIX commands, represented as UWL operators, appear in [5, 6]. Currently, our softbot is familiar with over one hundred UNIX commands, enabling it to satisfy a wide range of UNIX goals.

2.2 A UWL Planner for UNIX

As [16] points out, a planner with incomplete information has three basic options: to derive conditional plans that are intended to work in all contingencies [5, 17, 18], 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.

Thus, we have chosen to interleave planning and information gathering in the UNIX domain. Our softbot employs SOCRATES, a partial-order planner that extends SNLP [2, 12] in several important ways. First, SOCRATES accepts UWL operators/goals and incomplete state descriptions. Second, SOCRATES interleaves planning with execution. Like BUMP [16], SOCRATES executes an action by ordering it before all unexecuted actions and updating its model of the world state. However, SOCRATES is also able to "backtrack over execution," inverting executed steps when necessary [11]. In addition, SOCRATES actually sends executed commands to the UNIX shell and updates its incomplete world model based on UNIX's response. For instance, after SOCRATES successfully executes the `ls` command,

[1] The dynamic nature of the UNIX domain (users logging in, new files being created, etc.) means that this assumption is violated in practice. Indeed, our implementation relaxes it in a number of ways, but we do not discuss this issue here.

[2] We abbreviate positive literals of the form `((P !x) . T)` to `(P !x)`.

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

it dynamically creates objects in its world model that represent the files it has just encountered. Finally, as explained below, SOCRATES employs special mechanisms to avoid redundant information gathering.

3 Redundant Information Gathering

Since SOCRATES's world model is radically incomplete, much of its time is spent gathering information. We would like to be sure that SOCRATES does not do redundant work, pursuing information that it already possesses. As it turns out, SOCRATES can save itself a substantial amount of work by keeping track of what it knows and acting accordingly.

Consider, for example, the goal of determining whether the file `IJCAI-81.tex` is stored at a remote archive. The softbot can satisfy this goal by accessing the archive and searching it. However, doing so will waste a substantial amount of time if the softbot has recently searched the archive and recorded its contents. Merely observing that some information about the archive's contents is known does *not* necessarily obviate searching the archive again. Perhaps the softbot only recorded *some* of the files in the archive during its previous search. Unless the softbot has some means of concluding that it has *complete information* regarding the archive's contents, it will be forced to search the archive again to avoid missing any files.

The problem of redundant information gathering is general. For completeness, standard planning algorithms have to consider all methods of satisfying a goal. Whatever search strategy is used, each method is eventually tried until one succeeds or the goal is marked as unsatisfiable. Similarly, to ensure that an information goal will be satisfied, a planner has to consider all informative actions. Yet, once the information relevant to the goal is *known*, all informative actions are redundant. To avoid redundant information gathering an agent needs to "know what it knows." It requires a function that maps from information goals to T, indicating that all the relevant information is in its world model, or F indicating it is not.

Let K be a function from propositions to boolean truth values. K is true, for the proposition p, when all the propositions p_i, that are both true in the world and unify with p, appear in the agent's world model. When this is the case, we say that the agent has complete information regarding p. Otherwise, $K(p) = F$. Let W represent the set of propositions that are true in the world at a given instant, and let M represent the set of propositions that are true in the model at the same instant. Then, $K(p) = T$ is equivalent to the following:

$\forall p_i \in W$ such that *unifies*(p_i, p) we have that $p_i \in M$

Based on the value of K, the agent can decide whether to support an information goal with informative actions, or whether to rely exclusively on its model to satisfy the goal.

How can the agent determine whether it has complete information regarding a proposition? In general, answering this question could involve arbitrary reasoning regarding what the agent knows to be true, what it knows to be false, and the implications of these beliefs. However, our motivation for computing K is pragmatic: we would like to improve the performance of the agent. Thus, having it engage in arbitrary reasoning to compute K would be self-defeating. The agent requires an approximation to K, call it K', that can be computed *quickly*. If $K'(p) = T$ but $K(p)$ is false, for some proposition p, the agent could fail to satisfy a satisfiable goal. Therefore, the agent requires a conservative approximation to K where:

$$\forall p, K'(p) = T \quad \Rightarrow \quad K(p) = T$$

3.1 Computing K'

We have identified several classes of propositions for which K can be computed efficiently. Below, we consider each class in turn. The agent faithfully computes K for any proposition in these "tractable" classes, and returns $K(p) = F$ for any proposition p that is not. This is the agent's K'.

When the proposition p is ground, computing K' is trivial. If the agent's model contains p, or its negation, it follows that the agent has complete information regarding p. A similar outcome holds when p is not ground, but the values of the unbound variables in p are guaranteed to be unique.[4] This occurs when p encodes a functional relationship. Consider, for instance, the proposition `(word.count ?file ?count)`, which encodes a function from files to their word counts. It follows that when the file argument is bound, the word count has a unique value. Thus, if a UNIX softbot's information goal is:

`(find-out (word.count paper.tex ?count))`

And the softbot's model contains a proposition such as: `(word.count paper.tex 5000)` It follows that the softbot has complete information regarding the proposition in the goal. Functional relationships abound in physical domains as well as in the UNIX domain (most file attributes, such as length, owner, and so on, are functions). Making functional relationships explicit enables the UNIX softbot to efficiently determine that $K(p) = T$ in many cases.

When a proposition p encodes a function from its bound arguments to unique bindings for its unbound arguments (`?count` in the above example), the presence of a *single* instance of p, in the softbot's model, indicates that $K(p) = T$. This idea is easily generalized to the case where the unbound variables have multiple values, but the *number* of values is known. The softbot need only *count* the instances of p in its model to decide whether it has complete information regarding p (A similar idea is used to infer set closure

[4] This special case is handled by [1, 16].

in SERF [20].) However, many variables cannot be restricted in this way. Consider the proposition:

`(parent.dir ?file /bin)`

The variable `?file` can take on any number of values. Thus, the softbot *cannot* conclude that it has complete information regarding the files in `/bin` by *counting* the files it knows about. Although the above proposition can be viewed as a function from directories to *sets* of files, the cardinality of the sets is unknown. Thus, the softbot cannot tell whether it is "aware" of every element in the set. Another common case where variables have an unrestricted number of values is negative propositions such as: `(not (word.count paper.tex ?count))` An arbitrary number of instances of this proposition can be present in the model, but the softbot will not have complete information.

The softbot is aided, here, by informative actions that guarantee complete information regarding sets of values. For instance, executing `'ls -a /bin'` will inform the softbot about every file in `/bin`. The softbot annotates its model appropriately, and records that it has complete information regarding the proposition: `(parent.dir ?file /bin)`. To explicitly represent actions that yield complete information, we extend UWL by introducing a limited form of universal quantification in operator postconditions:

(∀ `!var s.t.` (< type >*)(< postcond >*))

This extension enables UWL to represent a run-time variable that is bound to *multiple* values as a result of sensing. For instance, the postconditions of `ls` are shown below.[5]

```
(∀ !file s.t. ((parent.dir !file ?dir))
   ((observe (parent.dir !file ?dir))
    (observe (name !file !name))
    (observe (pathname !file !path))))
```

When an operator is executed, its postconditions are used to update the world model. Each universally-quantified postcondition is translated into a set of ground propositions. In the case of `ls`, for example, if there are ten files in `?dir`, then thirty ground propositions are added to the model.

Our current representation does not explicitly denote when the operator provides complete information. However, it should be clear that `ls` provides complete information regarding the files whose `parent.dir` is `?dir`, since the `observe` postcondition exhausts `!file`'s type specification. In general, complete information is obtained when the type of a universally-quantified variable is identical to the content of one of the postconditions, within the scope of the universal quantifier. The softbot records this fact in its model and uses it to compute K'.

[5]Although implemented, we are still searching for the best syntax for this extension, and trying to precisely state its semantics. We are indebted to Steve Hanks and Dan Weld for extensive discussions on both counts.

In each of the cases described above, the softbot is able to efficiently and correctly compute K. The union of the cases constitutes K'—the softbot's approximation to K. In other cases, the softbot simply returns $K'(p) = F$. Our primary motivation for computing K' is to avoid redundant information gathering. We note, in passing, that K' is also useful in satisfying universally-quantified goals. For instance, if the softbot wants to compress all files whose size exceeds 100 megabytes, it may need to identify *all* such files and compress each file individually. To guarantee that it "knows" all the relevant files, the softbot requires a function like K'.

3.2 Uninformative Actions

K alone is not sufficient to avoid redundant information gathering. Consider a goal of the form: `(find-out (name ?file paper.tex))`. The softbot will not have complete information regarding the proposition mentioned in this goal, since it does not know the names of all the files in its universe. However, it ought to know that particular actions will fail to provide it with any new information. For instance, if the softbot is familiar with each file in the `/bin` directory, then the command `'ls -a /bin'` is *uninformative*. To avoid uninformative actions, the softbot requires a function, denoted by U, that maps an action and a proposition to T, indicating that the action is uninformative relative to p, or to F indicating that the action is informative.

The functions K and U have a simple relationship that is worth noting briefly. Complete information regarding a proposition p implies that no action is informative relative to that proposition. That is:

$$K(p) = T \Rightarrow \forall a, U(a,p) = T$$

Furthermore, if an informative action exists, then $K(p) = F$. The converse is false; lacking complete information regarding p does *not* imply that an informative action exists.

Again, computing U may involve arbitrarily complex reasoning, so the softbot relies on U', a conservative but efficient approximation to U. To compute U', the softbot records when it has obtained complete information regarding a set of objects, even if it does not have complete information regarding the proposition involved. For example, since the softbot cannot be familiar with all the files accessible through the Internet, it cannot have complete information regarding propositions such as `(name ?file paper.tex)`. However, after executing `'ls -a /bin'`, the softbot records that it knows the `name` and `pathname` of every file in `/bin`. thus, it is able to determine that:

U'`(ls(-a,/bin),(name ?file paper.tex))=T`

and avoid redundant uses of `ls`. In general, the softbot computes $U'(a,p)$ by checking whether the postcondition of a, that unifies with p, yields information that is already in the model. If so, the softbot declines to use the action a even when $K'(p) = F$.

4 Conclusion

The UNIX domain has led us to investigate the problem of planning with incomplete information. We have formulated the UWL representation (in collaboration with a number of colleagues [5]), developed an algorithm that interleaves planning and execution, and introduced the notion of locally complete information. A domain shapes and directs one's research in many ways, providing a source of intuitions, motivating examples, simplifying assumptions, stumbling blocks, test cases, etc. However, the results we describe are general. UWL is a general-purpose language for planning with incomplete information (e.g., [19] relies on UWL to specify diagnosis and repair actions for physical artifacts). Likewise, the notion of locally complete information, introduced in this paper, applies to a wide range of domains and a host of planning algorithms. Hence, we believe that UNIX, and real-world software environments in general, provide not only pragmatically convenient testbeds, but also a useful focus for planning research.

Our next step is to tackle the dynamic nature of the UNIX domain. SOCRATES periodically encounters execution failures because, contrary to SOCRATES's world model, one of the executed action's preconditions is not satisfied. For example, SOCRATES will attempt to initiate a talk session with a user who logged out long ago. We have extended UWL to include a **sense** goal annotation; a **sense** goal can only be satisfied by executing an informative action. When SOCRATES detects the failure of an action, it will attempt to test whether the action's preconditions are satisfied, and update its model accordingly. If a precondition turns out to be unsatisfied, SOCRATES has the option of trying to achieve the precondition, or choosing an alternative action to satisfy its goal. Appropriately handling out of date information, execution errors, and exogenous events, are general problems whose study is facilitated by the UNIX softbots testbed.

References

[1] J. Ambros-Ingerson and S. Steel. Integrating Planning, Execution, and Monitoring. In *Proceedings of AAAI-88*, pages 735–740, 1988.

[2] A. Barrett and D. Weld. Partial Order Planning: Evaluating Possible Efficiency Gains. Technical Report 92-05-01, University of Washington, Department of Computer Science and Engineering, July 1992.

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

[4] M. Drummond. A representation of action and belief for automatic planning systems. In *Proceedings of the 1986 workshop on Reasoning about Actions and Plans* , San Mateo, CA, 1986. Morgan Kaufmann.

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

[6] O. Etzioni, N. Lesh, and R. Segal. Building softbots for UNIX. Technical report, University of Washington, 1993.

[7] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4), 1971.

[8] J. Hintikka. *Semantics for Propositional Attitudes.* Cornell University Press, 1962.

[9] L. Kleinrock. Distributed systems. *Computer*, 18:90–103, November 1985.

[10] P. Langley and M. Drummond. Toward an experimental science of planning. In K. P. Sycara, editor, *Proceedings of the workshop on innovative approaches to planning, scheduling, and control.* Morgran Kaufmann, 1990.

[11] N. Lesh. A planner for a unix softbot. Internal report, 1992.

[12] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of AAAI-91*, pages 634–639, July 1991.

[13] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[14] R. Moore. A Formal Theory of Knowledge and Action. In *Formal Theories of the Commonsense World.* Ablex, 1985.

[15] L. Morgenstern. *Foundations of a Logic of Knowledge, Action, and Communication.* PhD thesis, New York University, 1988.

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

[17] M. A. Peot and D. E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on AI Planning Systems*, June 1992.

[18] M. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of IJCAI-87*, pages 1039–1046, August 1987.

[19] Y. Sun and D. Weld. Beyond Simple Observation: Planning to Diagnose. In *Proceedings of the 3nd International Workshop on Principles of Diagnosis*, pages 67–75, October 1992.

[20] M. Wellman and R. Simmons. Mechanisms for Reasoning about Sets. In *Proceedings of AAAI-88*, pages 398–402, August 1988.

[21] R. S. Wurman. *Information Anxiety.* Doubleday, 1989.