Partial Observability, Quantification, and Iteration for Planning
Work in Progress

Robert P. Goldman
SIFT, LLC
rpgoldman@sift.info

Abstract

"Look up all of the books for a course, and order each of them." This seemingly simple task cannot be planned by any planner that we know of. Classical planners are unable to handle this problem because it involves information-gathering, partial observability, and a domain of individuals that is not known at plan time. While substantial progress has been made in planning under incomplete information, this kind of goal remains out of reach, because even contingent planners can't handle sensing actions that produce set-valued results, nor can they operate over sets of unknown cardinality.

In this paper, we present an approach for planning that handles gathering information about sets of entities, and can operate on these sets. Our approach extends previous work on knowledge-based planning under partial information by adding reasoning about sets of entities whose membership and cardinality will be discovered only at runtime. This allows us to model limited forms of iteration and branching in the solution plans, handled by recursive planner invocation. We describe Window SHOP, a successor to SHOP2 currently under construction that incorporates our approach. While Window SHOP is an HTN planner, the approach described here is one of augmenting the state representation and progression algorithms of a planner, so is readily adaptable to first-principles planners.

Introduction

Classical planners make assumptions of complete observability and closed worlds that are not tenable in domains like web service composition (Wu et al. 2003), and "softbots" (Etzioni 1993). In these domains, the agent cannot just know the state of the world; it must plan to use information-gathering actions, and must track its state of knowledge. These domains also have an open domain of quantification. For example, a softbot interacting with a conventional workstation may create or delete an arbitrary number of files. Similarly, for an agent interacting with web services for retail the set of possible products, in addition to being unknown a priori, is effectively infinite. Agents in these environments will not be able to use the tactic of the universal base[1] for dealing with universally quantified goals, as conventional planners do.

The PUCCINI(Golden 1998) and PKS (Petrick and Bacchus 2004) planners deal with the problem of partial observability and open domains by using *local closed world* (LCW) information. Conventional planners use the closed world assumption: they track the state of the world as a set of ground literals, and if they want to know if some ground literal, l holds, they can simply examine their state database: if l is present, it is true, and if l is absent, they can treat it as false. In open, partially observable domains, the closed world assumption is not tenable, but full open world reasoning is terribly expensive. Accordingly, both PUCCINI and PKS use techniques that couple a conventional state database with a database of information about the limits of the agent's knowledge (the LCW database). When trying to determine whether l holds, such a planner will first examine its database for l. If l is present, it holds. If l is absent, then the agent consults the LCW. If l falls in a domain about which the agent has LCW, then a closed world assumption applies, and the agent knows it is false; otherwise, it is simply unknown. For example, if a softbot uses the `ls` command to list the contents of a directory, it will then have LCW of all of the files in that directory. Now, if it wishes to know if the file `shop2.lisp` is in that directory, it can consult its database, and if it does not see an assertion to that effect, it knows that `shop2.lisp` is not there.

In addition to using LCW, these planners also have the ability to construct plans that contain *run-time variables* (skolem functions). These run-time variables allow the planners to refer to information that will only be available to them at run-time. For example, if we are building a plan to order a book through the web, we might have a web service that, given an ISBN number, will return a price. So we could build a plan that first looks up the price for the book, checks to see if the price is below a threshold for acceptability, and if so, executes a credit card transaction. Note that a conventional planner could not construct such a plan, because it would need to instantiate the price at plan time and check again at plan time to see if the price was acceptable.

We address a critical limitation that these planners share: the inability to reason about sets of entities that will be known only at run-time. So, for example, they could generate the plan we refer to above, to buy a certain book from a web service. However, they would be unable to generate a plan for a goal like: "Consult the website for my algorithms course, find the list of textbooks, and order all of the textbooks listed there." There are three reasons for this limitation: first, the planners' run-time variables are constrained to refer only to individuals, and not sets; second, the LCW update algorithms are not capable of handling updates of unknown cardinality; and third, the planners cannot generate plans that contain loops. PUCCINI worked around some of these limitations by interleaving planning and execution

[1] Translating a universally quantified formula into a finite conjunction of ground formulas.

and by a special-purpose technique for linking to universally quantified goals (Golden 1997).

This paper addresses those three limitations: we introduce extensions to the LCW reasoning schemes that permit reasoning about run-time information about sets of objects that satisfy a particular description. We also introduce planning methods that introduce iteration constructs that operate on those sets. We are currently working on an implemnetation of these ideas, Window SHOP, built on the ReSHOP planner (successor to SHOP2(Nau et al. 2003)). While ReSHOP is an HTN planner, nothing here is especially particular to HTN planning. We concern ourselves with the state representation of the planner, and how this state representation can be used to plan for limited cases of iteration. The results should be readily applicable to first-principles planners, as well.

Actions

We model actions in our domain using a modification of PDDL, influenced by Golden's SADL(Golden 1997) and by description logics (Baader et al. 2003). Our actions have preconditions (which may be complex), like PDDL actions, but their postconditions are marked as either *causal* or *observational*. Causal effects may be conditional and are written as follows:

(cause (forall *vars* (when *cond* *effect*)))

The universally-quantification and the `when`-condition are optional, giving unconditional ground efects as the degenerate case. Observation effects are of the following form:

(observe [*dvar*][2] *description* [*attributes*])

Note that some previous systems have categorized entire actions, rather than just individual effects, as either causal or observational. Also note that the degenerate case of the observation effect is the ground observation, as in the operator of checking to see if the light is on in a room.

A sample operator, representing the Unix `ls -l` command, is given as Figure 1. If we want to know the contents of a directory with a particular pathname,[3] we can execute the action `(ls-l pth dir)`. As a result of this action, we will know all the elements of `dir`. Additionally, for each of those elements, we will know whether or not it is a file or a directory. We will know the pathname of each element, and its filesize. Finally, we will know that each element's parent-directory is `dir`.

```
(action ls-l
  :parameters (pth - pathname dir - dir)
  :preconditions (pathname dir pth)
  :effects
  (observe x
    (in.dir x dir)
    (and
      (observe (file x))
      (observe (directory x))
      (observe y (pathname x y))
      (observe y (file-size x y))
      (observe (parent-dir x dir)))))
```

Figure 1: Sample action, formalization of the Unix command `ls -l`.

Epistemic State

In order to handle the software and web-service domains of interest to us, Window SHOP must be able to reason about its own partial knowledge, and use information-gathering actions that it can use to expand that knowledge. Window SHOP avoids full belief-space planning, instead favoring a simpler incomplete, but sound representation of knowledge state, based on the idea of *local closed world (LCW) knowledge*(Etzioni, Golden, and Weld 1997). We have extended previous work on LCW knowledge to cover limited cases of reasoning about sets of individuals. We describe the components of this knowledge state, and a method for progressing the state over observations and actions.

Local closed world knowledge is a mechanism for overcoming the limitations of the closed world assumption (CWA) in domains where the classical planning assumptions do not hold, without resorting to full first-order theorem-proving. The CWA is not compatible with problems like "Order all of the books for Algorithms 221," if the planner does not know the list of books for the course. A CWA planner will simply say that there are no books for the course, because it doesn't have any its KB, and call it a day. Etzioni, *et al.*(1997) developed the technique of LCW as a way of indicating for what propositions the CWA is valid. For example, before reading the course syllabus, we don't have LCW, so when we query the set of all the books for Algorithms 221, the answer is "unknown," not false. After reading the syllabus, the CWA is valid, and if the agent doesn' know any such books, then it can conclude there are none. Etzioni, *et al.* provide algorithms for updating LCW over observations. Unfortunately, these algorithms do not provide a full answer for updating LCW *projectively*, in advance of the actual observations. That means that in order to handle problems like the textbook problem, PUCCINI must interleave planning and execution, and must block planning until observations are actually performed. We provide projective techniques for planning with observations here.

Golden's PUCCINI handled operators like the one in Figure 1, but because it did not handle sets, was unable to projectively plan tasks that involved such operators. Instead it would plan as far as it could without information, and would then stop planning and execute observations. For many situations, this is fine, but in other cases we might want to generate so-called "universal plans," generate plans for an interpreter that does not integrate planning, observation actions might be expensive or change the state of the world, etc.

[2]The *dvar* is only needed if additional attributes are specified.

[3]The entities are modeled separately from their pathnames to permit renaming, etc.

State components

PUCCINI's state representation is divided into two components:

1. A set of ground propositions, as in a conventional planner;
2. A set of LCW statements, which are universally-quantified conjunctions.

For example, if we were to list the directory `/Users/rpg/papers/`, the resulting PUCCINI state might be as given in Table 1. The LCW assertions indicate that the agent has complete knowledge of matching facts. So, for example, the first LCW assertion is intended to indicate:

```
(forall x
  (or (know (in.dir x dp))
      (know (not (in.dir x dp)))))
```

It's worth noting here that this state could only be computed after actually *executing* the list action; we cannot *project* these effects. Note also that knowledge reflected by the LCW assertions gets, in a sense, *weaker* as the number of conjuncts increases. For example, knowing all of the *files* in the directory (line 3 of the LCW in Table 1) does not entail knowing all of the entities in the directory (line 1). Golden (1997) provides algorithms for conservatively updating the LCW database over the state updates. These algorithms are efficient and sound, but not complete, in that they may remove LCW assertions unnecessarily in some circumstances. Golden provides experimental results, based on the softbot domain, that show that the incompleteness is not problematic.

Let us consider what we need to do to extend this framework to projective reasoning about information-gathering. Our information-gathering actions will produce information about the sets of entities that match some description. For example, all of the books that are to be read for a given class (consulting a syllabus) or all of the entities in a directory (using the `ls` command). This suggests incorporating in the agent's epistemic state a component that represent the sets whose membership that agent know, an epistemic state component we refer to as **Knowset**. We do this using a variant of the universally-quantified conjunctive representation used by PUCCINI for LCW. The representation differs in that it permits only a single universally quantified variable, and denotes the elements of the set in the description. We are grateful to Ugur Kuter for pointing out that the conjunctive representation may also be viewed as a description logic concept definition. So, in the state following listing the directory `dp`, the **Knowset** state component would include `(in.dir` x `dp)`[4]. An advantage of this scheme is that we can carry over Golden's LCW-updating algorithms to our **Knowset** case almost unchanged.

The LCW assertions of PUCCINI are used simply to check whether or not a proposition that is not in the agent's belief state can be assumed false. The **Knowset** facts, on the other hand, specify sets whose memebership is known, and about whose members we wish to predicate additional facts. Accordingly, the elements of the **Knowset** must be indexed in some ways, so that we can provide additional facts about them. We may learn some ancillary facts about the elements of these sets. Typically this will happen because of operating on one of these sets. So with every set in **Knowset**, we store propositions that we know to hold about the elements of the set. In this case we would have `(compressed` s`)` (where the variable s is implicitly quantified over the members of the set). Note that if we know all of the files that satisfy some description D, and we compress them, then we know additionally the set of files that satisfy $D \wedge$compressed. However, we know more than these two facts; we also know that $D \rightarrow$ compressed (and thus that there are no non-compressed files in D). In the same way as unary predications, we may record property-value assertions about the set elements.

When projecting the effects of observations, there will also be cases where we will need to record the fact that the agent *will know* at run-time whether or not a property holds of the members of a set (although we do not know at plan-time). For example, after we execute the `ls -l` action, we will not only know all of the entries in the directory, but also whether they are (sub)directories or normal files. So with every set in **Knowset**, we store propositions about which we have "knowif" knowledge, in this case `(file` s`)` and `(directory` s`)` (where the variable s is implicitly quantified over the members of the set). Finally, we may have "knowval" knowledge, where we know the value of some property (function) of the elements of a set. E.g., the `ls -l` action will also supply our agent with "knowval" knowledge of `(size` s`)`.

To summarize, given a known set whose description is P, we may have:

- Plan-time knowledge about whether a proposition ϕ holds of the elements: $\forall(s)P(s) \implies K\phi(s)$.
- Plan-time knowledge about the value of a property, f of the elements: for some k, $P\forall(s)(s) \implies Kf(s) = k$.
- knowif(ϕ) for some proposition ϕ, free in s:

$$\text{knowif}(\phi) \equiv \forall(s)P(s) \implies (K\phi(s) \vee K\neg\phi(s)) \quad (1)$$

- knowval(f) for properties, f:

$$\text{knowif}(\phi) \equiv$$
$$\forall(s)P(s) \implies \forall(x)\,(Kf(s) = x \vee Kf(s) \neq x) \quad (2)$$

Note that for both knowif() and knowval() there's a degenerate case of no free variables in the expression. For example, knowif(`(compressed f1)`) and knowval(`(size f1)`).

Initial-state modality

We also follow Etzioni, *et al.* in providing an initial state modality (`init`). This is necessary to appropriately capture goals that are framed in terms of descriptions of entities. For example, in the Unix domain, a user might want to print all the postscript files in a directory. That user would presumably be (unpleasantly) surprised if the system moved *additional* files into the directory and printed them as well, or if

[4] From now on, free variables are implicitly universally-quantified.

Ground propositions	LCW
(pathname dp "/Users/rpg/papers")	$\forall(x)$ (parent-dir x dp)
(parent-dir d1 dp)	$\forall(x,y)$ (and (parent-dir x dp) (pathname x y))
(parent-dir d2 dp)	$\forall(x)$ (and (parent-dir x dp) (file x))
(parent-dir f1 dp)	$\forall(x)$ (and (parent-dir x dp) (directory x))
(pathname d1 "/Users/rpg/papers/icaps09")	
(pathname d2 "/Users/rpg/papers/workshop")	
(pathname f1 "/Users/rpg/papers/.DS_store")	
(file f1)	
(directory dp)	
(directory d1)	
(directory d2)	

Table 1: Example of PUCCINI's state.

if *cond* known true then
    add *literal* and LCW(*literal*)
else if knowif($cond$) then †
    add knowif($literal$)
else if *cond* unknown then
    Make literal unkown...

Figure 2: Updating state with a causal effect

it was to simply delete all the postscript files in that directory and say "look, nothing to be printed!" That goal may be expressed as something like:

```
(forall (f dir)
  (init (pathname dir "/Users/rpg/papers"))
  (init (in.dir f dir))
  (printed f))
```

State update

The state of the agent must be updated to reflect the *projected* results of executing actions. Operators are either causal or observational, and has an effect expression which is a conjunction of individual causal or observational effects, respectively.

Causal effects Causal effects are of the form `(when` *cond* `(cause` *ex-literal*`))`, where the only free variables in *cond* are bound by the operator's parameters, and *ex-literal* is either a ground literal, or an existentially-quantified literal, which will be skolemized. To update the state with such an effect, Window SHOP follows the procedure given in Figure 2. With the exception of the line marked †, this procedure is exactly as per Etzioni, *et al.*, and inherits their soundness. The †line is needed because Window SHOP may need to reason about a state in which the executing agent *will* know whether *cond* is or is not true, but for which the planning agent cannot project a truth value; its correctness should be obvious.

Observational effects The interpretation of observational effects is sketched in Figure 3. Observational effects are centered around the optional description. The first branch of the if handles the simple case where there is no description. Here we simply record that the agent knows if e holds.

if ground(e) then
    unless $s \vDash e \vee \neg e$
        add knowif(s, e)
else
    ;; there is a description
    let s = **Knowset**(desc); †
    foreach c in attributes ‡
        if ground(c)
            add know(s, c)
        elsif obs(c)
            ;; c is a sub-observation...
            if quantified(c)
                add knowval(s, c)
            else
                add knowif(s, c)

Figure 3: Updating state, s, with observational effect, e.

We point out that this knowledge may be subsumed by more complete knowledge of the status of e. We suppress this detail in the rest of Figure 3. If there is a description present, then we must find the corresponding set (see the line with the †) — Window SHOP may already know such a set, or we may need to allocate a new one. The new one must be situated in the context of other sets because it may either inherit information from previously-existing subsumers or supply information to previously-known sets that it subsumes. Once we have the set in hand, we populate it with additional information (‡).

Planning with branches and loops

Our approach to epistemic planning with branches and loops is based on recursive invocations of the planner. When planning for a branch, conditional on ϕ the planner will split its state into two child states, plan recursively, and then merge the states together and continue. To plan to apply a task to a set, the planner will construct a new state, containing information about a representative element of the set, plan the task recursively for this element, while creating and enforcing loop invariants, and then abstract the resulting plan into a loop. We discuss these techniques in greater detail below.

ReSHOP

For modeling structured knowledge about a problem domain, one of the best-known approaches is *Hierarchical Task Network (HTN)* planning. An HTN planner formulates a plan by decomposing tasks (i.e., symbolic representations of activities to be performed) into smaller and smaller subtasks until tasks are reached that can be performed directly. The basic idea was developed in the mid-70s (Sacerdoti 1975; Tate 1977), and the formal underpinnings were developed in the mid-90s (Erol, Hendler, and Nau 1994). SHOP2 (Nau et al. 2003) is a modern, high-performance, open-source HTN planner. ReSHOP is a rearchitecting of SHOP2, based on object-oriented facilities provided by CLOS, in order to make it easier to tailor and extend SHOP2. It is used as the basis of a PDDL durative-action version(Goldman 2006), and a conditional planning version(Kuter et al. 2007).

In SHOP2, the domain-specific knowledge is encoded by means of *tasks* (i.e., symbolic representations of real-world activities), and *task-decomposition methods* that specify the possible ways of decomposing those tasks into smaller ones. Planning starts with a task network, specifying the tasks to be achieved, and constraints on the order in which they should be achieved. SHOP2 proceeds by decomposing those tasks into smaller tasks until only primitive tasks that can directly be executed in the world is left. The primitive tasks along with their ordering constraints constitute a solution plan for the input planning problem.

SHOP2 uses a search-control strategy called *ordered task decomposition*: the planner chooses to decompose tasks into subtasks in the same order that the tasks are supposed to be accomplished. As a consequence, SHOP2 generates the steps of each plan in the same order that the plan executor will execute those steps. Because of this search strategy, SHOP2 knows the current state at each step of the planning process. Since it is easier to reason about what is true than what might be true, this makes it easy to incorporate substantial expressive power into the planning system, such as the auxiliary functions and axioms mentioned earlier.

Information gathering

Window SHOP, like Golden's PUCCINI, can plan for goals of the form `(forall var (implies (description var) (goal var)))`. That is, achieve *goal* (or perform a task) for all entities that satisfy *description*. As with PUCCINI, Window SHOP will decompose this task by attempting to achieve knowledge about *description*, and then achieve the goal for the entities that satisfy the description. Like PUCCINI, Window SHOP will plan for the goal directly if it can determine the membership of *description* at plan-time. However, we go beyond PUCCINI and PKS in attempting to establish that the agent *will* know the membership of the description at some point in the plan, and can use iteration to establish the goal propositions will hold of all elements of this set.

Conditionals

The first of the two extensions to Window SHOP's planning is the added ability to plan conditional branch goals of the form `(if prop then-task else-task)`. To do this, Window SHOP creates two new world state objects, one for the case where *prop* holds, and one for the case where its negation holds, and then recursively plans *then-task* and *else-task* starting in those two states. If both of these recursive planning tasks succeed, then Window SHOP merges the two final states projected in the recursive planning processes, and continues planning any remaining tasks from the new, merged state. Note that this method uses the HTN planning facilities to generate conditional plans that look like conventional programs, with embedded branches, as distinguished from simpler conditional planners, such as Plinth (Goldman and Boddy 1994), which created tree-shaped plans, whose size would explode in the presence of multiple contingencies, and from planners like Cond-SHOP2(Kuter et al. 2007) that generate policies.

Window SHOP's algorithm for merging states after a branch is a conservative, sound but incomplete operation. Recall the components of Window SHOP state: ground assertions, set information and LCW. To merge states Φ and Ψ into Φ', we do the following:

1. Standardize the name of runtime variables and sets, so that they agree across Φ and Ψ.
2. For each $l \in \Phi_g$, if $l \in \Psi_g$, $l \in \Phi'$ else if $\Psi \models \neg l, \mathrm{knowif}(l) \in \Phi'$.
3. For each $\mathrm{knowif}(l) \in \Phi_g$, if $\mathrm{knowif}(l) \in \Psi_g$ then $\mathrm{knowif}(l) \in \Phi'$.
4. For each set descriptor, d in Φ and Ψ, $d \in \Phi'$.
5. For each $E \in Phi$ of the form $\forall x \in \mathcal{S}, \mathrm{know}(\mathcal{Z}(x))$, where $\mathcal{Z}(x)$ is of the form $\mathcal{P}(x)$, $\neg\mathcal{P}(x)$, $\mathcal{R}(x,c)$ $\neg\mathcal{R}(x,c)$, if $E \in \Psi$ then $Ein\Psi'$ else if $\Psi \models \forall x \in \mathcal{S}, \mathrm{know}(\neg\mathcal{Z}(x))$ add $\mathrm{knowif}(\mathrm{pos}(E))$ where $\mathrm{knowif}(\mathrm{pos}(E))$ is the positive form of E.
6. For each $E \in \Phi$ of the form $\forall x \in \mathcal{S}, \mathrm{knowif}(\mathcal{Z}(x))$, if $E \in \Psi$, $E \in \Phi'$.
7. For each $E \in Phi$ of the form $\mathrm{knowval}(\mathcal{R}, \mathcal{S})$, if $E \in \Psi$ then $E \in \Psi'$.
8. For each LCW conjunction, $L \in \Phi$, if $L \in \Psi$, then $L \in \Phi'$.
9. There are no other elements in Ψ'.

Iteration

Window SHOP provides the ability to plan a limited form of iteration, specifically "foreach" iteration over the elements of a set, for example `(foreach ?file in Set22 (print ?file))`. Its general technique is to perform a kind of universal instantiation on the set, synthesizing a planner state that contains facts about a representative element of the set. Window SHOP then invokes itself recursively to plan the task – in our example, `(print ?file)` – for the representative element of the set. The recursive planning is slightly modified from normal in order to establish and maintain loop invariants. If the recursive planning completes successfully, Window SHOP performs universal generalization on the resulting state, and continues planning.

Instantiating a loop variable Recall that the Window SHOP state contains four kinds of information about sets: a set description, know(), knowif() and knowval(). When we wish to loop over the set of members of a state, we perform the equivalent of universal instantiation using this knowledge. First, we create a unique individual, and instantiate the set description with this individual. So, for example, if we know the membership of the set of all files in my home directory, arbitrarily named `Set22`, `Set22 = (forall ?x (and (file ?x) (parent.dir ?x /home/rpg/)))`, Window SHOP might create `(and (file file0) (parent.dir file0 /home/rpg/))`. Next, Window SHOP uses the same new individual to instantiate the know() and knowif() expressions. The knowval() expressions are also instantiated, but require an additional instantiation for the slot-fillers. For example, `(forall ?x in Set22 (knowval (size ?x)))`, must be instantiated to `(size file0 size1)`. Two additional steps must be made: first, the facts about the set (`Set22` here) must be removed from the new state, and second, the LCW database must be updated appropriately. In the above example, `(LCW (size file0 size1))` and `(LCW (size file0 ?x))`[5] will be added to the LCW state. The task, of course, must be instantiated in parallel: `(print file0)`.

Recursive planning and loop invariants After instantiating the new state, Window SHOP proceeds to plan for the task in that new state. However, there is one difference between this planning and conventional ReSHOP planning. Here, since Window SHOP is planning a loop body, it must be sure to enforce loop invariants. In order for each step in the recursive plan to be valid, all of its preconditions must be satisfied. Each precondition may be satisfied in one of two ways: either (1) that precondition is satisfied in the initial state of the loop or (2) that precondition is established by an earlier loop step. If the precondition is satisfied in the initial state of the plan, Window SHOP will *protect* the condition until the end of the loop body. If the precondition is established by an earlier loop step, no protection is needed, because in each iteration of the loop it will be established anew.

Two points must be made about the above technique: First, the statement here is oversimplified. The technique described in the previous paragraph will only work in the absence of conditional effects of the form `(when cond effect)`. If a conditional effect is used to establish a precondition, then the condition for that effect, *cond*, must be protected, and so must any of its preconditions. Second, the requirement here is actually too strict. If a precondition, p, is consumed by a, it is actually acceptable that p be clobbered after a, *so long as it is reestablished by the end of the loop body*. At the moment, we have not implemented this alternative; it's not clear to us whether it is of practical importance in our domains.

[5]Because there is only one size for each file.

Generalizing the loop plan If Window SHOP successfully plans the loop body, then it must be generalized and inserted in the plan so that Window SHOP can plan any remaining tasks. The generalization is the inverse of the operation we have described above: when Window SHOP establishes a proposition about the iteration variable, it is lifted to apply to the full set in the state after the loop. Note that this may require existential generalization as well as universal generalization if in the course of the loop Window SHOP will come to learn an attribute of the iteration variable. Note also that the loop *may* clobber knowledge of the set over which it iterates; the semantics of the loop involves iteration over the set elements *at the time the loop commences*. In practice this does not happen in our problems, and it would not create problems for the semantics of loop execution. However, it would complicate handling goals of the form `(forall var (implies description goal))`; it is for this reason that Golden introduced the **initially** modality.

Related Work

In the introductory material we have already discussed some of the most directly related work, notably that by Golden, *et al.* and by Petrick.

Hoffman, *et al.*(Hoffmann et al. 2009) discuss a technique for doing web service composition using conformant planning. Their work aims to address issues of entity creation and of integrity constraints over the individuals in the domain. Their approach differs from ours in that they stay within the now-standard framework of grounded PDDL-style planning. The relevance of the work is that they identify a notion of "forward-effects" (and strictly forward effects) that enable them to work in a propositional framework while accommodating entity construction. We believe that the same notion may be used to tame some of the difficulties of reasoning with the witness variables introduced in planning loops in our approach.

Levesque's work on planning with loops (Levesque 2005) is similar to our own in aiming to efficiently handle a subset of possible uses of loops in plans. However, Levesque chooses a different class of loops as his focus — those that are required by the existence of a planning parameter that is not bounded. His paradigm cases is of a tree that can be felled by some number of chops from an axe, where the number of chops is not known by the planning agent. Our paradigm case is similar in that we are forced to use a loop only because we do not a priori know the membership of the set(s) of interest. However, our approach is more one of generating a mapper than a loop, and ion general there is no single action that could handle an arbitrary element of the set in the way the chop action handles an arbitrary "chop-resistance" of the tree.

Levesque's work on planning with loops is related to previous work on cyclic planning. Most recently, Cimatti, et al. have characterized various different types of cyclic planning (Cimatti et al. 2003), but such work dates back at least 15 years (Musliner 1994).

Srivastava *et al.* (Srivastava, Immerman, and Zilberstein 2008) provide a method for generalizing a plan for some set

of objects into a looping plan that abstracts away from a particular set of objects. This is similar to our work in its use of a plan for representative elements, and may be more general in the looping constructs it permits; our loops are limited to being essentially mappers. Our work differs in aiming to handle entities that are unknown a priori rather than generalizing from known entities to arbitrary sets. It would be interesting to compare their abstraction techinque with our witness-based approach, and see if it provides means to generalize the loops or provide better characterizations of the kinds of plans this technique can and cannot find.

Conclusions

This paper describes work in progress. We have designed modifications to the ReSHOP data structures needed to capture the epistemic states described here. We have partial implementations of the projection algorithms. We are investigating whether algorithms used for the most restrictive classes of description logic will provide valuable assistance in this. The greatest practical difficulties involve recognizing when knowledge may be *destroyed* by an action (e.g., by possibly changing the membership of a knownset), so that we retain soundness, without being so conservative as to be useless. At the same time, we are examining a number of test domains to determine whether the limitations on our input language are appropriate. We are also working on a treatment of creation of new entities, which we need to handle tasks like file creation for softbots and the creation of reservations, etc. for web service applications. Finally, although we believe that in many cases preplanning is critical, we do not reject interleaving execution and planning in all cases.

Acknowledgments

This article was supported by DARPA/IPTO and the Air Force Research Labratory, Wright Labs under contract number FA8650-06-C-7606. Approved for Public Release, Distribution Unlimited. This paper does not represent the official position or opinions of DARPA/IPTO or Air Force Research Labratory, Wright Labs.

Our thanks to Keith Golden for helpful discussions, and for providing us with his formalization of the Unix softbot domain. Thanks also to Ugur Kuter, Ron Petrick, John Maraist, John Phelps, Mark Burstein, Drew McDermott, Dana Nau, and the members of the University of Maryland, College Park SHOP research group for helpful discussions. Thanks to Mike Pelican for discussions of LCW implementations and help with copyediting.

References

American Assoc. of Artificial Intelligence. 1994. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Menlo Park, CA: AAAI Press/MIT Press.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P., eds. 2003. *The Description Logic Handbook — Theory, Implementation and Applications*. Cambridge University Press.

Cimatti, A.; Pistore, A.; M, R.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1–2):35–84.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), 1123–1128.

Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence* 89(1–2):113–148.

Etzioni, O. 1993. Intelligence without robots: A reply to Brooks. *AI Magazine* 14(4):7–13.

Golden, K. 1997. *Planning and knowledge representation for softbots*. Ph.D. Dissertation, University of Washington.

Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, 70–77. Menlo Park, CA: AAAI Press.

Goldman, R. P., and Boddy, M. S. 1994. Conditional linear planning. In Hammond, K. J., ed., *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference*. Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Goldman, R. P. 2006. Durative planning in HTNs. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 382–385.

Hoffmann, J.; Bertoli, P.; Helmert, M.; and Pistore, M. 2009. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research* 35:49–117.

Kuter, U.; Nau, D.; Reisner, E.; and Goldman, R. P. 2007. Conditionalization: Adapting forward-chaining planners to partially observable environments. In *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems*.

Levesque, H. J. 2005. Planning with loops. In Kaelbling, L. P., and Saffiotti, A., eds., *IJCAI*, 509–515. Professional Book Center.

Musliner, D. J. 1994. Using abstraction and nondeterminism to plan reaction loops. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), 1036–1041.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Petrick, R. P. A., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, 2–11.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, volume 4, 206–214.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In Fox, D., and Gomes, C. P., eds., *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, 991–997. AAAI Press.

Tate, A. 1977. Generating project networks. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, 888–893.

Wu, D.; Parsia, B.; Sirin, E.; Hendler, J. A.; and Nau, D. S. 2003. Automating DAML-S web services composition using SHOP2. In *International Semantic Web Conference*, 195–210.