Acting on information: A plan language for manipulating data

Keith Golden
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
kgolden@ptolemy.arc.nasa.gov

Abstract

Information manipulation is the creation of new information based on existing information sources. This paper discusses problems that arise when planning for information manipulation and introduces an action language, ADLIM, that addresses these problems. Topics include

- How to represent information in a way sufficient to express the effects of actions that modify the information.
- How to concisely represent actions that copy information, or produce new information that is based on existing information sources.
- How to generate a pipeline of information-processing commands that will produce an output containing the desired information.

1 Introduction

A big part of NASA's job is information management. Satellites, unmanned spacecraft, planetary rovers and observatories, for all their complexity, can all be viewed as remote sensors; their sole purpose is to gather data, which they transmit across a vast, interplanetary network, where it is processed, delivered to the scientific community, and archived. Just as controlling spacecraft is a complex task that can benefit from automation, so is managing the data that spacecraft returns. Currently, automation usually consists of writing some scripts to handle expected cases. Some problems with this approach are lack of flexibility in the face of changing requirements or one-time requests, lack of robustness in the face of errors or unanticipated events, and poor data tracking. The latter is a serious problem, since much of the information about various data files and how they are inter-related is stored in the heads of operators, who will eventually go on to other missions, or forget.

We are developing a system, called IMAGEbot[1], to automate information manipulation tasks, such as generating mosaics or converting file formats, and to automatically record in a database the vital meta-information about files that have been produced, such as how they were generated, or which parts of the telemetry logs correspond to which science data. IMAGEbot will robustly respond to errors, such as disks filling up, and will be able to provide greater functionality than hand-generated scripts, since it will generate its own scripts based on user-supplied goals.

This work builds on experience with the Internet Softbot (Etzioni & Weld 1994) and other information-gathering agents. However, the information manipulation problem is beyond the grasp of current softbots. There has been considerable work on developing planners for *gathering* information, but almost no work on planners for *manipulating* information. That is not because manipulating information is not important, but because representing actions that manipulate information is hard. For example, in the Internet Softbot, the copy action was never cleanly or accurately represented, and it was impossible to represent Unix pipes (which redirect the output of one command to the input of another) or actions like `tar` (which creates an archive of a collection of files) and difficult to achieve image processing tasks.

We present an action language, ADLIM, that is capable of representing such actions easily and concisely. ADLIM stands for Action Description Language for Information Manipulation, and is an extension of Pednault's ADL (Pednault 1989). It is intended for problem domains, such as space science missions, where information manipulation tasks, like image processing and data archiving and distribution, are commonplace.

The next section discusses how actions and goals for information manipulation are represented in ADLIM. Section 3 discusses a problem that comes up when trying to represent actions that copy or modify information and shows how it is a generalization of the frame problem. Section 4 discusses temporal projection in ADLIM and illustrates it with a simple example from a planetary rover domain. Section 5 discusses related work.

2 Information manipulation

Since the purpose of ADLIM is to represent actions that process data inputs and produce data outputs, inputs

[1]IMAGE stands for Information Manipulation, Archiving, Gathering and Extraction

and outputs are explicitly declared in action descriptions. Every variable is declared as an input, output, parameter or quantified variable. Inputs and outputs are distinct from other variables in that an input is not guaranteed to exist after the action is executed, and an output does not exist before the action is executed.

An action that has no inputs and one or more outputs is called an *information source.* An action that has one or more inputs and no outputs is an *information sink.* An action that has one or more inputs and one or more outputs is an *information filter.* An information filter processes the inputs, producing the outputs. Contrary to the behavior of a physical filter, it does not necessarily remove anything from the input, and may add something or change it completely.

Effects

UWL (Etzioni *et al.* 1992) and SADL (Golden & Weld 1996) represent information-producing actions using the annotation **observe**. For example, to represent that executing the action `ls /bin` reveals the name of every file in `/bin`, the SADL encoding would be:

$$\forall f\, \exists n\ \textbf{when}\ (\mathsf{in.dir}(f,\ \texttt{/bin}))\ \textbf{observe}(\mathsf{name}(f,\ n))$$

However, the encoding using the **observe** annotation does not actually represent the effects of `ls`, but rather the combination of `ls` with a program to interpret its output and produce a set of knowledge base updates. Such a program is called a *wrapper.* Since a filter works on the syntactic output of a program, not the semantic interpretation, using **observe** throws away vital information that a planner needs to reason about the effect of a filter: the syntax. Instead, we describe the syntax of the output directly. We divide conditions that can be sensed into two categories: simple observables and fluents.

Simple observables are functions or relations whose value is independent of the situation in which they are evaluated. For example, the relation contains(`"foobar"`, `"foo"`) is easily answerable by examining only the syntactic representation of `"foobar"` and `"foo"`. Rather than providing a sensing action for a simple observable, we provide a function that returns its value (if all arguments are bound), or provides additional constraints for its arguments (if some arguments are not bound).

Fluents are functions or relations whose value depends on the situation. For example, the relation in.dir(`foo`,`/bin`), which means the file `foo` is in the directory `/bin`, will be true in some situations and false in others. A sensing action can be described using conditional effects, where the antecedent refers to the situation in which the action is executed, described using fluents, and the consequent refers to the contents of the output, described using simple observables. E.g., the effect of `ls` discussed above would be

$$\forall f\ (\mathsf{parent.dir}(f)\ =\ \texttt{/bin}) \rightarrow \mathsf{contains\text{-}line}(\mathsf{name}(f),\ out)$$

This translates to "For each file in directory `/bin`, there is a line in the output that is equal to the name of the file." The $\rightarrow$ is used instead of **when** (which is used in SADL) to indicate a conditional effect. The reason for this notation will become clear in the next subsection. We will use LHS to designate the expression on the left hand side of the $\rightarrow$ and RHS to denote the expression to the right of the $\rightarrow$. Note that we use function composition as a shorthand. The above goal is equivalent to

$$\forall f, n\quad (\mathsf{parent.dir}(f)\ =\ \texttt{/bin} \wedge n\ =\ \mathsf{name}(f)) \rightarrow \mathsf{contains\text{-}line}(n,\ out)$$

Two things to note are that name(f) is now on the LHS and n is universally quantified ($\exists$ was used in the SADL example). $\forall$ and $\exists$ are equivalent when the variable represents the value of a function, since there is alway exactly one possible value. We use $\forall$ here because it's simpler. Other examples in the paper can be transformed in the same way: replace each nested function with a new universally quantified variable and add an equality constraint between the variable and the function to the LHS.

Goals

Information manipulation goals, like effects, must be explicit about the syntax of the output. Consider the goal of outputting the result of a query to a file. Merely ensuring that the file contains the information is not sufficient. For example, a list of file names should not be mixed up with a list of userids, since it may be difficult to tell which is which. Furthermore, if the file is to be read by another program, there will be exact formatting requirements.

Suppose our goal is to to produce a killfile, which is a list of email addresses that we don't want to receive email from, each on a separate line. Let's say we don't want to receive email from anyone who has sent us email containing the string `"MAKE MONEY FAST."` We might express this goal as:

$$\forall em\ \mathsf{email\text{-}received}(em)\ \wedge\ \mathsf{contains}(\mathsf{subject}(em),\ \texttt{"MAKE MONEY FAST"}) \rightarrow \mathsf{contains\text{-}line}(\mathsf{sender}(em),\ \texttt{killfile})$$

This has the same form as the conditional effects discussed in Section 2, but the meaning is slightly different. Whereas, in effects, the statement $A \rightarrow B$ indicates that if A is true before the *action* is executed then B will be true after the *action* is executed, in goals it means that if A is true before the *plan* is executed, then B must be true after the *plan* is executed. In SADL goals, A would be represented using **initially** and B would be expressed using **satisfy** (Golden & Weld 1996). One problem with the above goal is that including all known email addresses in the killfile would also satisfy the goal. To express more restrictive goals, we also allow expressions of the form $A \overset{\leftrightarrows}{\rightarrow} B$, which is equivalent to $(A \rightarrow B) \wedge (\neg A \rightarrow \neg B)$.

Information pipelines

An information pipeline is one or more information sources followed by a sequence of filters, possibly terminated by an information sink, in which the output of each action is directed to the input of a following action. The effects of an information pipeline can be represented in the same way as the effects an individual action. The LHS refers to the conditions that are true when the first action is executed and the RHS refers to the contents of the output and the conditions that are true after the last action is executed. This representation can be generated automatically from the individual action descriptions, using the same process that is used in planning (see Section 4). In this manner, a precise description of of the contents of the output of a pipeline is generated, relating the meaning of the information to its form. This description can be archived, along with the time and circumstances under which the output was generated, for later use by IMAGEbot. Thus, if IMAGEbot has a goal, or subgoal, of generating a file with particular properties, and it has generated such a file before, it can use the previously generated file instead.

The same approach can be used to describe files or other information resources not generated by IMAGEbot. It can then use these as information sources for in its plans.

3 The copy problem

The representation of information filters presents a challenge. A filter creates a new object, such as a text file, which is based on an existing object. Although the input and output of a filter are distinct objects, they have much in common. The output may be a copy of the input, with some changes.

We call this the *copy problem,* which can be understood by considering the effects of a photocopier. Initially, we have an original of a document and a blank sheet of paper. After running the photocopier, everything that was true about the original is still true (the frame problem). Additionally, most everything that was true about the original is now also true about the formerly blank page (the copy problem). In some ways, the copy will differ from the original – for instance, it may be of lower quality, or the original may be color whereas the copy is black and white, but if the original contained, say, a tax form, or the first 500 digits of π, then so will the copy. Listing all of these conditional effects explicitly would be impractical.

The copy problem is just a generalization of the frame problem; if the original and the copy are the same object, then we have a restatement of the frame problem. But we are interested in the case where the objects are different.

There have been many solutions to the frame problem, from the STRIPS assumption to logical formalisms such as (Reiter 1991). None of these directly meets our needs, since they are solving a different problem. What we need is to *explicitly* state that an output is identical to an input unless stated otherwise. For example, when creating a compressed copy of a file, one should be able to declare that the size and compression of the files are different, but in all other respects they are identical. We refer to such declarations as *generalized frame effects* (GFEs): The effect $\mathsf{frame}(source, dest)$ in an action a means that, for any predicate or function p, the value of $p(dest)$ after a is executed is the same as the value of $p(source)$ before a is executed, unless contradicted by another effect of a. *Source* and *dest* need not be the same type, but predicates defined for *source* must also be defined for *dest*. Given such declarations, it is straightforward to formalize them using one of the elegant formal solutions to the frame problem.

We require that each *dest* be a newly created object and the target of only one GFE. However, a single output can comprise many distinct objects, each of which may be the target of a separate GFE (see the description of `tarc` below). Given incomplete information, not all the consequences of the GFE will be known, but anything known about *source* will be known about *dest*, and all prior positive knowledge will still be valid. However, negative knowledge, including LCW knowledge, may be invalidated.

One might imagine dispensing with the inputs and outputs and representing all actions as destructive. Then the STRIPS assumption could be used to preserve attributes of the files that don't change. This would require that all files that need to be preserved be explicitly copied (Chien *et al.* 1997). However, doing so merely pushes the frame problem into the action $\mathsf{copy}(f_1, f_2)$, since for any proposition $p(f_1)$ that is true before the copy, $p(f_2)$ should be true afterward. Furthermore, this approach is only applicable in cases where a single input is mapped to a single output. It will not help when modeling the effects of an action that generates mosaics (combining many images into one).

Example

To illustrate the use of frame effects, consider the Unix command `tar`, for creating and extracting tar files. A tar file is an archive (**tar** stands for **t**ape **ar**chive) of some collection of files. It is really little more that a concatenation of all the files, with some extra information to indicate where one file begins and another ends and the relative pathnames of the files. We refer to the portion of a tar file representing a particular file as a file record, and we use the predicate $\mathsf{contains}(t, r)$ to indicate that tar file t contains file record r. There is no magic in $\mathsf{contains}$. In fact, it's a simple observable, since, given the complete contents of a tar file and a file record, testing whether the file $\mathsf{contains}$ the record is a matter of a linear search through the tar file for contents that exactly match the record.

We represent the two functions of the `tar` command with two different actions. The `tarc` action *creates* a tar file from the contents of a directory, descending the directory hierarchy recursively. To avoid representing the recursion explicitly, tarc does not refer to directories, but to pathnames. A file is (recursively) contained within a directory if the pathname of the directory is a prefix of its own pathname. The parent.dir function can also be defined in terms of pathname. The output of tarc is a newly created tar file, containing a file record for each file reachable from the directory. Each file record is identical to the original file, except that it has only a relative pathname, and it is not located on any machine.

```
action tarc (path $dp$, exec-context $ec$)
 output: tarfile $out$
 precond:  pathname(currentdir($ec$)) = $dp$
 effect:
   $\forall$ file($f$),path($lp$)
       (pathname($f$) = concat($pd$,"/",$lp$) $\wedge$
       host-loc($f$) = currenthost($ec$))
         $\rightarrow \exists$ file-record($fr$)
              frame($f$, $fr$)$\wedge$
              host-loc($fr$) = nil $\wedge$
              contains($out$, $fr$)$\wedge$
              pathname($fr$) = $lp$
 exec: "tar cf -"
```

The tarx action *extracts* information from the tarfile and creates the corresponding files and directories in a new location. For each file record in the tarfile, a new file is created, identical to the file record, except that it has a new pathname and host location. Although these action descriptions omit some minor details, they are essentially complete. The key to their brevity is the frame effects, which stand for a huge number of statements.

```
action tarx (path $dp$, exec-context $ec_{in}$)
 input: tarfile $in$
 precond:  pathname(currentdir($ec_{in}$)) = $dp$
 effect:
   $\forall$ file-record($fr$),path($lp$)
      (pathname($fr$) = $lp$ $\wedge$ contains($in$, $fr$))
        $\rightarrow \exists$ file($f$)
             frame($fr$, $f$) $\wedge$ $\neg$contains($in$, $f$) $\wedge$
             host-loc($f$) = currenthost($ec_{in}$) $\wedge$
             pathname($f$) = concat($dp$,"/",$lp$)
 exec: "tar xf -"
```

It is common practice in Unix to use tar to copy large file hierarchies from one machine to another. In Section 4, we provide a short example showing how the frame effects allow properties to be preserved across copy operations using tar.

4 Reasoning about plans

Our definition of a planning problem is the standard one: Given descriptions of allowable actions, a goal and the initial state, produce a plan that, when executed starting from the initial state, will move the world to a state satisfying the goal. Regardless of the specifics of planning algorithm used, a planner following this definition must be able to answer one of the following questions:

- If I execute this action, how will the state change? (progression)
- If I want my goal to be satisfied after I execute this action, what needs to be true beforehand? (regression)

We discuss regression in ADLIM. We do not discuss progression, which is somewhat more complicated, but we note that since part of an ADLIM goal refers to the initial state, progression affects not just the initial state, but also the goal. We conjecture that under some circumstances (such as information gathering) that forward planning will be more efficient than backward planning, and that a mixed strategy is likely to work the best.

Regression

Goal regression means determining the conditions that need to be true in the initial state for an action or action sequence to achieve a given goal. We will use $\mathsf{R}_a(\Gamma)$ to represent the result of regressing Γ through action a, and $\mathsf{R}_{\{a\}_1^n}(\Gamma)$ to represent the result of regressing Γ through the action sequence $a_1; a_2; \ldots; a_n$. Regression closely follows (Pednault 1986).

Regression of the empty plan succeeds iff, in the initial state, the LHS implies the RHS.

$$\mathsf{R}_{\{\}}(\Phi \rightarrow \Psi) = (\Phi \Rightarrow \Psi)$$

Regression of a plan consists of successively regressing each action, starting with the last.

$$\mathsf{R}_{\{a\}_1^n}(\Gamma) = \mathsf{R}_{a_1}(\mathsf{R}_{a_2}(\ldots \mathsf{R}_{a_n}(\Gamma)))$$

Conjunction, disjunction, quantification and negation are handled in the usual manner. Namely, $\mathsf{R}_a(\Gamma_1 \wedge \Gamma_2) = \mathsf{R}_a(\Gamma_1) \wedge \mathsf{R}_a(\Gamma_2)$, $\mathsf{R}_a(\Gamma_1 \vee \Gamma_2) = \mathsf{R}_a(\Gamma_1) \vee \mathsf{R}_a(\Gamma_2)$, $\mathsf{R}_a(\neg\Gamma) = \neg\mathsf{R}_a(\Gamma)$, $\mathsf{R}_a(\forall x\Gamma) = \forall x \mathsf{R}_a(\Gamma)$ and $\mathsf{R}_a(\exists x\Gamma) = \exists x \mathsf{R}_a(\Gamma)$.

Given a goal of the form $\Phi \rightarrow \Psi$, regress the Ψ and leave the Φ alone (since it already refers to the initial state).

$$\mathsf{R}_a(\Phi \rightarrow \Psi) = \Phi \rightarrow \mathsf{R}_a(\Psi)$$

Finally, to regress a single literal: φ is true iff the action makes it true, or if it was true previously and the action doesn't make it false.

$$\mathsf{R}_a(\varphi) = (\Sigma_\varphi^a \vee (\varphi \wedge \Pi_\varphi^a))$$

where Σ_φ^a means action a enables φ (makes it true) and Π_φ^a means a preserves φ (doesn't make it false). Informally, $\Sigma_{p(c)}^a$ is true iff a has an effect $p(c)$, or if $p(o)$ is true, and a has an effect of the form frame(o, c) and doesn't have an effect $\neg p(c)$. Π_φ^a is equivalent to $\neg\Sigma_{\neg\phi}^a$.

Example

To illustrate goal regression, consider the following highly simplified example. Suppose our goal is to store the images downlinked from a planetary rover in the directory /images, with a compression quality of 95%. Achieving this goal requires no sensing.

$\forall$ image j $\exists$ file f, filename fn
fromdownlink(j)$\wedge$format(j)=JPEG $\rightarrow$
pathname(f) = concat("/images/", fn)$\wedge$
copy-quality(f, j)$\geq$ 0.95

We will regress this goal over "lifted" actions, as a planner would, but we gloss over certain subtleties of constraint reasoning, skolemization, and the like. The last action in the plan will be `tarx`, to extract files from a tar file, so we, regress through `tarx` first. We will call that action a_1, and rename all variables from the action by prefixing them with "a_1." The goal pathname(f) = concat("/images/", fn) is satisfied by a_1 if concat("/images/", fn) = concat($a_1.dp$, "/", $a_1.lp$). Note that one solution for this constraint is $a_1.dp$ = "/images". If we provide the background knowledge that fn, being a filename, cannot contain "/", and $a_1.dp$, being a pathname, cannot be the empty string, then that solution is unique. However, even without that background knowledge, $a_1.dp$ = "/images" is the only solution that will be found as part of a correct and unambiguous plan. Any other solution either leaves the suffix of parameter $a_1.dp$ unspecified or adds an unsupported constraint to the prefix of $a_1.lp$.

The predicate copy-quality does not appear in the effect of action a_1, but there is an effect frame($a_1.fr$, $a_1.f$), which will transfer the copy-quality of $a_1.fr$ to f. Thus, the goal copy-quality(f, j)$\geq$ 0.95 will be satisfied if copy-quality($a_1.fr_j$, j)$\geq$ 0.95 is true in the prior state, where $a_1.fr_j$ is a new existentially quantified variable, within the scope of j, that is used in place of the $\forall$ variable $a_1.fr$. Finally, the LHS of the conditional effect and the precondition of the action must be true. Thus, the goal is now:

fromdownlink(j)$\wedge$format(j)=JPEG $\rightarrow$
$a_1.dp$ = "/images" $\wedge$
pathname($a_1.fr_j$) = fn $\wedge$
contains($a_1.in$, $a_1.fr_j$) $\wedge$
copy-quality($a_1.fr_j$, j)$\geq$ 0.95 $\wedge$
currentdir($a_1.ec_{in}$) = $a_1.dp$.

Regressing through the `cd` action satisfies the currentdirectory goal. Then regress through `tarc`, which we will designate a_2. The contains condition will be satisfied if $a_1.in = a_2.out$. The pathname condition is satisfied if the LHS is true. Once again, the copy-quality condition can be satisfied by resorting to the frame effect of a_2, resulting in the goal copy-quality($a_2.f_j$, j)$\geq$ 0.95. The resulting goal agenda is:

fromdownlink(j)$\wedge$format(j)=JPEG $\rightarrow$
$a_1.dp$ = "/images" $\wedge$ $a_1.in$ = $a_2.out$ $\wedge$
copy-quality($a_2.f_j$, j)$\geq$ 0.95 $\wedge$
pathname($a_2.f_j$) = concat($a_2.pd$,"/",fn) $\wedge$
host-loc($a_2.f_j$) = current.host($a_2.ec$) $\wedge$
currentdir($a_2.ec$) = $a_2.dp$

The constraint on $a_1.dp$ specifies a parameter of `tarc`, the directory to extract the tar file into. The constraint $a_1.in = a_2.out$ specifies that the output of `tarx` must be directed to the input of `tarc`. Once again, regressing through `cd` satisfies the currentdirectory goal. Now suppose there is an action `get-downlink`, which gets all the JPEG images from the downlink and stores them in a directory `/downlink/jpg`, ensuring that each image has a quality of 0.99. Call that action a_3. The effect of a_3 would be

fromdownlink($a_3.g$)$\wedge$format($a_3.g$)=JPEG $\rightarrow$
copy-quality($a_3.c$, $a_3.g$)= 0.99 $\wedge$
pathname($a_3.c$) =
concat("/downlink/jpg/", filename($a_3.c$) $\wedge$
host-loc($a_3.c$) = current.host($a_3.ec$)

The resulting goal agenda would be

fromdownlink(j)$\wedge$format(j)=JPEG $\rightarrow$
$a_1.dp$ = "/images" $\wedge$ $a_1.in$ = $a_2.out$ $\wedge$
$a_2.pd$ = "/downlink/jpg/" $\wedge$ $a_2.ec$ = $a_3.ec$ $\wedge$
fromdownlink($a_3.g$)$\wedge$format($a_3.g$)=JPEG

The last two conditions are implied by the LHS of the goal, so they are automatically satisfied. The remaining conditions are parameter choices, which are trivially satisfied. Thus, the plan `get-downlink;cd /downlink/jpg; tar xf - | '(cd /images; tar cf -)'` achieves the goal. In practice, downlinks are not initiated directly by the planner, and the goal would involve moving the files to a different machine rather than just a different directory. That could be accomplished by inserting an `rsh` (remote shell) command between the `tarx` and `cd` commands.

5 Conclusions

We presented ADLIM, the Action Description Language for Information Manipulation, which can concisely represent actions that copy all or part of an input to an output. We observed that this is a generalization of the frame problem, which has not been noted before in the planning literature, and presented a solution, using frame effects. We showed how information gathering can be accomplished in this framework.

Future work

We are building a constraint-based planner for ADLIM, but many issues still need to be resolved. Constraint reasoning in ADLIM is especially challenging, since most constraint solvers assume that variable domains are static, whereas, in ADLIM, domains may be completely or partially unknown. We are working on a constraint solver, using procedural constraints (Jönsson 1997), that can cope with unknown domains.

Related work

We have already discussed the relation between ADLIM and UWL (Etzioni *et al.* 1992), ADL (Pednault 1989) and SADL (Golden & Weld 1996). Collage (Lansky & Philpot 1993) and MVP (Chien *et al.* 1997) both automate image manipulation tasks, a motivating problem for ADLIM. However, they don't focus as much on accurate causal models of information manipulation. MVP requires actions to destructively modify their inputs, relying on the STRIPS assumption to preserve properties not listed in the action's effects. Collage relies solely on abstract action decomposition and thus does not need a precise causal theory of the actions.

Representing actions that manipulate information is related to representing sensing actions. (Moore 1985) introduced a theory of knowledge and action, based on a variant of the situation calculus with possible-worlds semantics, which included an analysis of information-providing effects. (Scherl & Levesque 1993) built on Moore's work, providing a solution to the frame problem for knowledge-producing actions. The semantics for ADLIM has been specified following their formalization, but we base our language on ADL, which allows for more tractable planning. Our treatment of knowledge-producing actions using conditional effects follows (Pryor & Collins 1996) and others, but we are unaware of work that treats goals in a similar manner.

There are many other action languages that represent sensing, but none of them have the expressiveness of ADLIM. They either disallow sensing the value of a variable (Levesque 1996; Goldman & Boddy 1996; Pryor & Collins 1996), thus restricting sensors to returning a finite set of possible values, or they disallow the use of conditional effects to describe sensing actions (Kwok & Weld 1996; Levy, Rajaraman, & Ordille 1996; Knoblock 1996; Babaian & Schmolze 1999; Etzioni *et al.* 1992), which is essential for representing information outputs that can be manipulated by other actions.

Acknowledgements

I would like to thank Jeremy Frank, Ari Jonsson, David Smith, Ellen Spertus, Richard Washington and the anonymous reviewer for helpful comments.

References

Babaian, T., and Schmolze, J. G. 1999. PSIPLAN: Planning with ψ-forms over partially closed worlds. Unpublished.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining.*

Etzioni, O., and Weld, D. 1994. A softbot-based interface to the Internet. *C. ACM* 37(7):72–6.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, 115–125.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

Goldman, R. P., and Boddy, M. S. 1996. Expressive Planning And Explicit Knowledge. In *Proc. 3rd Intl. Conf. AI Planning Systems.*

Jönsson, A. 1997. *Procedural Reasoning in Constraint Satisfaction.* Ph.D. Dissertation, Department of Computer Science, Stanford University.

Knoblock, C. 1996. Building a planner for information gathering: A report from the trenches. In *Proc. 3rd Intl. Conf. AI Planning Systems.*

Kwok, C., and Weld, D. 1996. Planning to gather information. In *Proc. 13th Nat. Conf. AI.*

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93).*

Levesque, H. 1996. What is planning in the presence of sensing? In *Proc. 13th Nat. Conf. AI.*

Levy, A. Y.; Rajaraman, A.; and Ordille, J. J. 1996. Query answering algorithms for information agents. In *Proc. 13th Nat. Conf. AI.*

Moore, R. 1985. A Formal Theory of Knowledge and Action. In Hobbs, J., and Moore, R., eds., *Formal Theories of the Commonsense World.* Ablex.

Pednault, E. 1986. *Toward a Mathematical Theory of Plan Synthesis.* Ph.D. Dissertation, Stanford University.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. Principles of Knowledge Representation and Reasoning*, 324–332.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research.*

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy.* Academic Press. 359–380.

Scherl, R., and Levesque, H. 1993. The frame problem and knowledge producing actions. In *Proc. 11th Nat. Conf. AI*, 689–695.