Acting on information: Representing actions that manipulate information

Topic: Representational Formalisms (Action)

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

November 1, 1999

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 proposes a novel action representation, called ADLIM, that addresses these problems, including:

- How to represent information in a way sufficient to express the effects of actions that modify the information. I present a simple, yet expressive, representation of information goals and effects that generalizes earlier work on representing sensing actions.
- How to concisely represent actions that copy information, or produce new information that is based on existing information sources. I show how this is a generalization of the frame problem, and present a solution based on *generalized frame effects*.
- How to generate a pipeline of information-processing commands that will produce an output containing exactly the desired information. I present a new approach to goal regression.

1 Introduction

There are many problem domains that consist largely of actions that *manipulate* information. For example, image processing tasks consist of taking one or more input images, transforming them in various ways, and writing the result to one or more output files. Network administration also involves substantial information manipulation, such as backing up files, applying patches to software, compiling and installing programs, modifying configuration files.

This is the sort of task that planner-based softbots[1] [4] were designed for. 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 unimportant, but because representing actions that manipulate information is hard.

This work builds on experience with the Internet Softbot [4] and on current efforts to build softbots for information manipulation tasks at NASA. Years of building a domain theory for the Internet Softbot made clear the representation language's strengths and weaknesses, which led to a number of improvements, as discussed in [6, 8, 7]. One area where it has remained weak, indeed where all action languages are weak, is representing the information content of files. The copy action was never cleanly or accurately represented, and we were unable to represent Unix pipes (which redirect the output of one command to the input of another). Actions like `tar` (which creates an archive of a collection of files) were out of the question.

This paper discusses the reasons why it is difficult to represent such actions in previous action languages, and presents a new language, ADLIM, that addresses these problems. ADLIM stands for Action Description Language for Information Manipulation. ADLIM is based on Pednault's ADL [19], and strongly influenced by SADL (Sensory ADL). [6] Whereas SADL focuses on gathering information, ADLIM focuses on manipulating information.

1.1 Roadmap

Section 2 introduces the ADLIM action language, defining the semantics of effects and goals in terms of the situation calculus. Section 3 discusses the problem that occurs when information is copied, shows that this problem is

[1] **Softbot** stands for **software robot**.

a generalization of the frame problem, and introduces a new kind of effect to deal with it. Temporal projection is fundamental to planning. Section 4 discusses temporal projection in ADLIM, providing a sound and complete goal regression operator. The full paper will also discuss a new technique called goal *progression*, which ADLIM's unique representation allows. The full paper will also discuss the relationship between information manipulation and information gathering, show how the latter is handled in ADLIM, and discuss how a planner can reason about the completeness of information contained in a file, using Local Closed World reasoning [3]. Section 5 provides a brief example of ADLIM actions and Section 6 discusses related work.

2 Information manipulation in ADLIM

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

2.1 Background: The Situation Calculus

I define the semantics of ADLIM expressions by mapping them into the situation calculus [16], a first-order logic used to describe changes to the world resulting from the execution of actions. A *situation* is a state of the world at a given time. A *fluent* is a function or proposition whose value may change over time. To indicate the value of fluent $\varphi(x)$ in situation s, I will write $\varphi(x, s)$. At any one time, there is some situation s that holds at that time, and for any boolean fluent $\varphi(x)$, either $\varphi(x, s)$ or $\neg\varphi(x, s)$. Thus, the set of ground facts holding in situation s comprises a complete logical theory.

However, if an agent has incomplete information about the world, it cannot know what the actual situation is, since that would require knowing the value of every fluent. Thus, there are many other situations that might hold instead. Following [21] and many others, I represent these situations using the predicate K. $K(s', s)$ is true if and only if it is consistent with an agent's knowledge in situation s to believe that the actual situation might be s'.

All changes to the world are assumed to be the result of executing actions. The special function *do* is used to represent these changes. $do(a, s)$ returns the situation that results from executing action a in situation s. $\{a\}_1^n$ represent a sequence of actions $a_1; a_2; \ldots; a_n$. $do(\{a\}_1^n, s)$ is equivalent to $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$, *i.e.,* the result of executing the entire sequence, starting in situation s. s_0 represents the initial situation.

Since ADLIM expressions are not themselves objects in the situation calculus, I introduce the following predicate, which takes ADLIM expressions as arguments. $HOLDS(E, \{a\}_1^n, s)$ means that the ADLIM expression E becomes true as a result of executing action sequence $\{a\}_1^n$ from situation s.

2.2 Effects

UWL [5] and SADL 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:

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

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 successfully create a data pipeline. Instead, I divide conditions that can be sensed into two categories:

- **simple observables** are functions or relations whose value is independent of the situation in which they are evaluated. An example of a simple observable is the relation contains("`foobar`", "`foo`"). It is easily answerable by examining only the syntactic representation of "`foobar`" and "`foo`". For every simple observable, there is a function that returns

its value (if all arguments are bound), or provides additional constraints for its arguments (if some arguments are bound). Formally, Ψ is a simple observable if for any variable substitution θ, either $\Psi\theta$ or $\neg\Psi\theta$ is true in all situations:

$$\forall\theta\,[\forall s(s \models \Psi\theta) \vee \forall s(s \models \neg\Psi\theta)]$$

- **contingent facts** 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 contingent facts, and the consequent refers to the contents of the output, described using simple observables. For example, the effect of `ls` discussed above would be represented as

  $\forall f$ (parent.dir(f) = `/bin`) $\rightarrow$ contains-line(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."[2] The $\rightarrow$ is used instead of **when** to indicate a conditional effect. The reason for this notation will become clear in Section 2.3.

Nested expressions like contains-line(name(f),*out*) in the above example are nice notationally but awkward for a planner to work with. They can be decomposed by introducing additional variables. For example, if n =name(f), then the above can be rewritten as contains-line(n,*out*). However, it is necessary to decide which side of the $\rightarrow$ the n =name(f) belongs on. That is, does n refer to the name of f before the action is executed or after? I adopt the simple rule that, by default, sub-expressions like name(f) always refer to the situation *before* the action is executed. The exceptions to this default will be spelled out in the full version of this paper.

Effect Semantics

The semantics of effects is mostly straightforward. I will explain the semantics by translation into the situation calculus. Let s_a be the situation before

[2]This describes the "piped" version of ls (which is appropriate in this context), without any flags. More complex versions of ls, such as ls -l, can also be easily represented.

the action a is executed and let s_b be the situation that results from executing action a in situation s_a ($s_b = do(a, s_a)$). Translation consists of converting the $\rightarrow$ into an implication ($\Rightarrow$) and specifying the situation that each expression refers to. Each fluent will acquire an additional argument referring to the situation. For example, name(f) becomes name(f, s). Fluents left of the $\rightarrow$ pertain to s_a and those on the right pertain to s_b. For example, the effect of `ls` discussed above translates to

$$\forall f \text{parent}.\text{dir}(f, s_a)\text{=/bin} \wedge n = \text{name}(f, s_a) \Rightarrow \text{contains-line}(n, out, s_b)$$

I sidestep the ramification problem by making the STRIPS assumption: Any proposition that is not explicitly made true or false by an action remains the same. The only exception is that if the value of a function is changed, it is not necessary to declare that the previous value no longer holds. For a formalization of the STRIPS assumption in the situation calculus, see [21].

Any effect can be rewritten so that each consequent appears only once. First, conjunction is eliminated in the consequent by replacing expressions of the form $\Phi \rightarrow \varphi_1 \wedge \varphi_2$ with $(\Phi \rightarrow \varphi_1) \wedge (\Phi \rightarrow \varphi_2)$. Then effects of the form $\Phi \rightarrow \varphi$ and $\Psi \rightarrow \varphi$ are combined into a single effect $(\Phi \vee \Psi) \rightarrow \varphi$. Disjunction is only allowed in the antecedent of effects, so after this transformation, the effects can be specified completely by listing the preconditions of each (possibly quantified) literal φ or $\neg\varphi$. For each literal φ, let $\gamma_\varphi^T(a)$ be the condition required for action a to make φ true, and let $\gamma_\varphi^F(a)$ be the condition needed to make φ false. If a does not affect φ, then γ_φ^T and γ_φ^F are both false. Since the effects of a must be consistent, $\neg\gamma_\varphi^T \vee \neg\gamma_\varphi^F$. For simplicity, I assume in this discussion that no effects cause φ to become unknown, but see [7] to see how that is handled. It then follows that

$$HOLDS(\gamma_\varphi^T \rightarrow \varphi, a, s) \Rightarrow (\gamma_\varphi^T(s) \Rightarrow \varphi(do(a, s)))$$

2.3 Goals

Information manipulation goals, like effects, must be explicit. 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 my goal is to to produce a killfile, which is a list of email addresses that I don't want to receive email from, each on a separate line. Let's say I don't want to receive email from anyone who has sent me a message containing the string "MAKE MONEY FAST." I might express this goal as:

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

This has the same form as the conditional effects discussed in Section 2.2, 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. Thus, A is similar to an **initially** goal in SADL and B is similar to a **satisfy** goal [6].

As was argued in [8, 6], information goals are inherently temporal. It doesn't make sense to ask about the value a fluent without specifying at what time the value of the fluent is to be sampled (and at what time the answer is desired). This notation assumes information goals are of the form "Tell me *ASAP* the value that these fluents have *now*" or, more generally, "Act *ASAP* on the *current* value this expression."

Goal Semantics

The structural similarity between goals and effects is not coincidental. The semantics of a goal is equivalent to the semantics of an effect, with the exception that the temporal extent of a goal is the entire plan, rather than just a single action. That is, s_a is replaced with s_0, the initial situation, and s_b, is replaced with $s_n = do(\{a\}_1^n, s_0)$.

The $\rightarrow$ in goals represents implication between a formula in s_0 and a formula in s_n, but often bi-implication is the desired interpretation. For example, consider the above example. Including the email address of every person I've received email from would satisfy the above goal, but is not at all what I want. To address this problem, a *strict* interpretation can be indicated by the notation $\overset{\Rightarrow}{\rightarrow}$. Using that notation, the above goal is equivalent to the following expression in the situation calculus.

$$\forall em, p \ \text{email-received}(em, s_0) \ \wedge \ p = \text{sender}(em, s_0) \ \wedge$$
$$\quad \text{contains}(\text{subject}(em, s_0), \ \text{"MAKE MONEY FAST"})$$
$$\quad\quad \Leftrightarrow \ \text{contains-line}(p, \ \texttt{killfile}, \ s_n)$$

In general,

$$HOLDS(\Phi \rightarrow \Psi, \{a\}_1^n, s_0) \equiv \Phi(s_0) \Rightarrow \Psi(do(\{a\}_1^n, s_0)$$
$$HOLDS(\Phi \stackrel{\Rightarrow}{\rightarrow} \Psi, \{a\}_1^n, s_0) \equiv \Phi(s_0) \Leftrightarrow \Psi(do(\{a\}_1^n, s_0)$$

3 A new frame 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 (possibly minor) changes. For example, `gzip`, a compression utility, leaves all properties of the file the same, with the small difference that the output is compressed and the size is different. Explicitly representing all of the ways in which the input and the output are the same would be cumbersome, since usually they will be more similar than different. For example, if the input of `gzip` contains a picture of the Martian rock named Yogi, then so does the output. If the input contains the postscript version of this paper, then so does the output. Listing all of these conditional effects explicitly would be impractical.

This is just a generalization of the frame problem, which is typically solved by the STRIPS assumption. The STRIPS assumption is not adequate to represent the copying that occurs in filters, since the input and output are distinct, and thus any propositions that refer to them will also be distinct. What is needed is the ability to explicitly state that the output is identical to the input unless stated otherwise. For example, in the case of `gzip`, one should be able to declare that the size and compression of the file are different, but in all other respects the files are identical. I refer to such declarations as *generalized frame effects*: The effect $\mathsf{frame}(x, y)$ in an action a will be used to mean that for any proposition $p(x)$ that holds before the action is executed, $p(y)$ will hold afterward, unless $p(y)$ is contradicted by another effect of the action.

Formally, this representation can be described in terms of a second-order logic. If R is the set of relations and F is the set of non-one-to-one functions, then $HOLDS(\mathsf{frame}(x, y), a, s))$ is equivalent to

$$\forall p \in R\,[p(x, s) \Leftrightarrow p(y, do(a, s)) \text{ unless } \gamma^T_{p(y)}(a, s) \vee \gamma^F_{p(y)}(a, s)]$$
$$\wedge \forall f \in F\,[f(x, s) = f(y, do(a, s)) \text{ unless } \exists z(\gamma^T_{f(y)=z}(a, s))]$$

which translates to "everything that is true for x before executing a is true for y afterward, unless a has some effect that specifies the value of the fluent in question." From this point onward, I will use frame(x, y, a, s) as a shorthand for $HOLDS(\text{frame}(x, y), a, s))$.

Note that one case has been left out. If $f(z, s) = x$, the frame declaration will not result in $f(z, do(a, s)) = y$. There is a good reason for this. Since f is a function, $f(z)$ can only have one value at a given time. Making $f(z) = y$ will result in $f(z) \neq x$, which is most likely unintended. The above definition has the advantage that the consequences of frame will be limited to the copy that is being made. If the copy is a newly created entity and all effects are limited to the copy, the value of previously true propositions will not be affected.

One might imagine dispensing with the inputs and outputs and representing all actions as destructive. Then the STRIPS assumption would preserve attributes of the files that don't change. This would require that all files that need to be preserved be explicitly copied [2]. However, doing so merely pushes the frame problem into the action 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), since many inputs are mapped to a single output.

4 Temporal projection

The structure of goals and effects has some interesting properties when it comes to planning. Since the goal contains a part that refers to the initial state and a part that refers to the final state, it is possible to simultaneously apply regression and progression to the goal, effectively working on it in two directions at once. Since the left side of the goal refers to the state of the world that the agent is to obtain information about and the right side refers to what is to be done with that information, there is no reason to assume that working on one side or the other will result in a smaller branching factor. A strategy such as least-cost flaw repair (LCFR) [10] could be used to choose at each iteration which side to work on.

Consider a goal of the form $A \rightarrow C$, and two actions, with effects $A \rightarrow B$ and $B \rightarrow C$, respectively. Regressing the goal through the second action

results in a goal of the form $A \rightarrow B$. Regressing that through the first action results in $A \rightarrow A$. Progressing the goal through the first action results in a goal of the form $B \rightarrow C$. Progressing that through the second action results in $C \rightarrow C$. In general, a planner can apply a series of regression and progression operators on the goal, until the left side implies the right side (and *vice versa*, in the case of $\stackrel{\Leftrightarrow}{\rightarrow}$).

4.1 Successor state axiom

Following [18], I introduce a *successor state axiom*, which determines the next state after executing action a, based on the previous state. First, I present some useful definitions. Following [18], I define the enabling conditions Σ^a_φ and the preservation conditions Π^a_φ of φ. The enabling conditions follow Pednault's definition, except that φ can also be enabled by frame effects.

$$\Sigma^a_{\varphi(y)}(s) \Leftrightarrow \gamma^T_{\varphi(y)}(a,s) \vee \left(\exists x(\varphi(x,s) \wedge \mathsf{frame}(y,x,a,s) \wedge \neg\gamma^F_{\varphi(y)}(a,s))\right)$$
$$\Sigma^a_{\neg\phi(y)}(s) \Leftrightarrow \gamma^F_{\varphi(y)}(a,s) \vee \left(\exists x(\neg\varphi(x,s) \wedge \mathsf{frame}(y,x,a,s) \wedge \neg\gamma^T_{\varphi(y)}(a,s))\right)$$

A condition is preserved as long as its negation is not enabled.

$$\Pi^a_{\varphi(y)}(s) \Leftrightarrow \neg\Sigma^a_{\neg\varphi(y)}$$
$$\Pi^a_{\neg\varphi(y)}(s) \Leftrightarrow \neg\Sigma^a_{\varphi(y)}$$

The Successor State Axiom states that if condition φ is true after executing action a, either a caused φ to become true, or φ was true originally, and a didn't cause φ to become false.

Theorem 4.1 *Successor State Axiom*
$\varphi(do(a,s)) \Leftrightarrow \Sigma^a_\varphi(s) \vee \varphi(s) \wedge \Pi^a_\varphi(s)$

4.2 Regression

Goal regression is used to determine what must be true prior to executing a sequence of actions to ensure that a given condition is true afterward. That is, if Γ is to be true in situation $do(\{a\}^n_1, s)$, what must hold in situation s?

First, I define regression of the empty plan, $\{\}$. Since the temporal extent is zero, regression succeeds iff the left side entails the right side

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

Regression of a non-empty 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 $\Phi \rightarrow \Psi$, we regress the Ψ and leave the Φ alone (since it already refers to the initial situation).

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

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

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

Regression in ADLIM is both sound and complete.

Theorem 4.2 *Soundness and Completeness of Goal Regression*

Let α be an axiomatization of the domain theory and the initial state. Then $\alpha \models (HOLDS(\mathsf{R}_{\{a\}_1^n}(\Gamma), \{\}, s_0) \Leftrightarrow HOLDS(\Gamma, \{a\}_1^n, s_0))$

There is insufficient space in this extended abstract to discuss progression of goals, but there will be a discussion of it in the full paper.

5 Example

The power of this representation can be seen when composing information-processing actions together. For example, in Unix, a common way of copying whole directory hierarchies is to create an archive of the files and execute a remote command to extract the archive on a target machine. This can be accomplished with a single command-line instruction, using the pipe operator "|" to redirect the output of one command into the input of the next:

```
tar cf - . | rsh target-host '(cd target-dir; tar xf -)'
```

The `tarc` action creates a tarfile 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.directory function can also be defined in terms of pathname. The output of tarc is a newly-created tarfile, 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
  effect: ∀ file(f),path(lp)
            (pathname(f) = concat(pd, lp) ∧
             host-loc(f) = currenthost(ec))
               → ∃ file-record(fr)
                     frame(f, fr)∧host-loc(fr) = nil
                     contains(out, fr)∧
                     pathname(fr) = lp
  exec: "tar cf - dp"
```

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 is the frame effects, which stand for a huge number of statements.

```
action tarx (path dp, exec-context ec)
  input: tarfile in
  precond: pathname(currentdirectory(ec_in)) = dp
  effect: ∀ file-record(fr),path(lp)
            (pathname(fr) = lp ∧ contains(in, fr))
               → ∃ file(f)
                     frame(fr, f) ∧
                     host-loc(f) = currenthost(ec) ∧
                     pathname(f) = concat(dp, lp)
  exec: "tar xf -"
```

The full paper will present a detailed example of how a planner can reason about this action sequence, using the temporal projection operators defined in Section 4.

6 Conclusion and related work

I presented ADLIM, an action description language for information manipulation, which is unique in the ability to concisely represent actions that copy all or part of an input to an output. Representing such actions presents a generalization of the frame problem, which has not been noted before in the planning literature. I presented a solution, using frame effects, and defined the semantics of the language in terms of the situation calculus.

Collage [13] and MVP [2] both automate image manipulation tasks, a motivating problem for ADLIM. However, they don't focus on accurately modeling 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 [17] devised 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. I opt for a less expressive language, for the sake of tractability. Scherl and Levesque [21] built on Moore's work, providing a solution to the frame problem and knowledge-producing actions. The semantics provided for ADLIM closely follows their formalization. Son and Baral [22] offer a simpler formalization.

ADLIM follows UWL [5] and SADL [6] in providing an action language suitable for softbots, but opts for a more general representation of sensing actions. ADLIM, like SADL, extends ADL [19], and adopts limited temporal quantification for information goals. However, whereas SADL'S sensing actions are expressed only partially in terms of conditional effects, ADLIM'S are expressed entirely using conditional effects and "simple observables".

There are many other action languages that represent sensing, such as [14, 9, 20], but none of them have the expressiveness of ADLIM. They either disallow sensing the value of a variable [14, 9, 20], thus restricting sensors to returning a finite set of possible values, or they disallow the use of conditional effects to describe sensing actions [12, 15, 11, 1, 5], which is essential for representing information outputs that can be manipulated by other actions.

References

[1] Tamara Babaian and James G. Schmolze. PSIPLAN: Planning with ψ-forms over partially closed worlds. Unpublished, 1999.

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

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

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

[5] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

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

[7] Keith Golden. *Planning and Knowledge Representation for Softbots.* PhD thesis, University of Washington, 1997. Available as UW CSE Tech Report 97-11-05.

[8] Keith Golden. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*, 1998.

[9] Robert P. Goldman and Mark S. Boddy. Expressive Planning And Explicit Knowledge. In *Proc. 3rd Intl. Conf. AI Planning Systems*, May 1996.

[10] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. 12th Nat. Conf. AI*, July 1994.

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

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

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

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

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

[16] 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.

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

[18] E. Pednault. *Toward a Mathematical Theory of Plan Synthesis.* PhD thesis, Stanford University, December 1986.

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

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

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

[22] Tran Cao Son and Chitta Baral. Formalizing sensing actions - a transition function based approach. Unpublished, 1998.