# Planning to Gather Information

Technical Report UW-CSE-96-01-04

Chung T. Kwok & Daniel S. Weld[1]
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{ctkwok, weld}@cs.washington.edu

## Abstract

The exponential growth of the Internet has produced a labyrinth of documents, databases and services. While almost any type of information is available somewhere, even expert users waste time and effort searching for appropriate information sources, and phrasing queries in the custom formats required by each site. To make matters worse, many queries can only be answered by combining information from several different sites.

This paper describes Occam, a query planning algorithm that determines the best way to integrate data from different sources. As input, Occam takes a library of site descriptions and a user query. As output, Occam automatically generates one or more plans that encode alternative ways to gather the requested information.

Occam has several important features: (1) it integrates both legacy systems and full relational databases with an efficient, domain-independent, query-planning algorithm, (2) it reasons about the capabilities of different information sources, (3) it handles partial goal satisfaction *i.e.*, gathers as much data as possible when it can't gather exactly all that the user requested, (4) it is both sound and complete, (5) it is efficient. We present empirical results demonstrating Occam's performance on a variety of information gathering tasks.

[1]We thank Paul Beame, Oren Etzioni, Marc Friedman, Keith Golden, Steve Hanks, Nick Kushmerick, Alon Levy, and Mike Williamson for helpful discussions. This research was funded in part by Office of Naval Research Grant N00014-94-1-0060, by National Science Foundation Grant IRI-9303461, by ARPA / Rome Labs grant F30602-95-1-0024, and by a gift from Rockwell International Palo Alto Research.

# 1 Introduction

The exponential growth of the Internet and World Wide Web has produced a labyrinth of documents, databases and services. Almost any type of information is available *somewhere*, but most users can't find it, and even expert users waste copious time and effort searching for appropriate information sources. Artificial intelligence and database researchers have addressed this problem by constructing integrated information gathering systems that automatically query multiple, relevant information sources to satisfy a user's information request [9, 5, 15, 18, 16, 29, 10]. These systems raise the level of the user interface, since they allow the user to specify *what* she is interested in without worrying about *where* it is stored or *how* to access the relevant sources [9].

These motivations inspire the Occam[2] planning system which we describe in this paper. Occam automates the process of locating relevant information sources from a repository of source *descriptions* and combining them appropriately to answer users' information requests. Unlike previous implemented systems, Occam integrates both legacy systems and full relational databases with an efficient, domain-independent, query-planning algorithm.

## 1.1 Assumptions

We assume that a network (*e.g.* the Internet) makes available numerous, distributed information sources with the following characteristics. Sources are managed by different organizations, hence agents (whether human or automated) must adhere to the remotely defined formats. The information services are potentially slow and expensive, so users must balance the cost of each access against its estimated benefit. The information sources are dynamic, hence an agent must recognize when a site's contents, protocol or performance changes, as well as when new sites come online and existing sites leave. Many sources represent *legacy systems* in the sense that they do not support a comprehensive query interface such as SQL; in these cases an agent may need to expend additional effort to determine the best way to

[2] William of Occam, 1285–1349, was an English scholastic philosopher best known for "Occam's razor," a rule in science and philosophy which states that the simplest of two or more competing theories is preferable. Occam is quoted as saying "It is vain to do with more what can be done with less." The Occam system embodies this philosophy by seeking the simplest plans that gather all information requested by the user.

answer an information gathering request.

## 1.2 A Simple Example

Suppose we want to find out the names of all people in an office. If we knew of a relational database containing this information, gathering the information would be easy. But suppose no such database exists. Instead we have only two information sources, namely, the UNIX *finger* command which returns the names of people given their email addresses, and the *userid-room* command which returns email addresses of all the occupants in an office. In order to find the names of the office's inhabitants, one must combine these sources: first issuing the *userid-room* command and then running *finger* on each of the email address returned. The Occam planner reasons about the capabilities of information sources (*e.g.*, legacy systems such as *finger*, *userid-room* as well as more powerful relational databases) in order to synthesize a sequence of commands that will gather the requested information. Since Occam realizes that information sources may not be exhaustive, when necessary it generates multiple plans in order to gather as much information as possible.

## 1.3 Context

In contrast to previous work from the AI planning community, Occam uses an action language that is designed to represent information sources; this enables a highly specialized planning algorithm. For example, the only preconditions to Occam operators are *knowledge preconditions* [21, 22, 8]. Furthermore, since the operators executed by Occam are requests to information sources, we need not model causal effects; hence, there are no *sibling-subgoal interactions* such as those characterizing the Sussman anomaly [26, 4]. Occam does not model the world state as do many other AI planners; instead it models the *information state*, which is a description of the information collected by Occam at a particular stage in planning.

In contrast to work on multidatabase systems, Occam provides a single unified world model that is independent from the conceptualization used by the information sources; this greatly simplifies integration of new sources. Moreover, Occam is more expressive than many multidatabase systems, since it is able to model the presence of incomplete information in sources. Unlike most multidatabase systems, Occam is equally adept at extracting information from both legacy systems and full relational databases.

## 1.4 Overview

We describe our representation of information sources and queries in the next section. Section 3 defines a plan and explains what it means for a plan to satisfy an information gathering request. The next two sections present the Occam planning algorithm: section 4 explains the generation of potentially useful action sequences and shows how Occam evaluates its version of the Modal Truth Criterion [4]. In section 5 we introduce two improvements to the algorithm and present an empirical evaluation of Occam's performance. The paper concludes with a discussion of related and future work.

# 2 Representing the World, Sites & Queries

Conceptually, Occam allows the user to interact with Internet services through a single, unified, relational database schema called the *world model*. For example, the world model might represent information about a person's email addresses with the *relation schema* `email(F,L,E)`, where `F, L,` and `E` represent the *attributes* firstname, lastname and email address respectively. Another relation schema `office(F,L,O)` could be used to record that the person with firstname `F` and lastname `L` has office `O`.

Occam is strongly typed. Occam associates a type with each variable and attribute; a variable may only appear as an attribute in a relation if the type matches. For example, within the `email(F,L,E)` relation, `F` and `L` are associated with the type *name*, and `E` is associated with *email*.

## 2.1 Representing Information-Producing Sites

We represent Internet services and other data sources by modeling the type of queries they are capable of handling and by specifying a mapping between their output and relations in the world model. Both purposes are achieved with *operators* which have two parts:

1. A *head* which consists of a predicate symbol denoting the *name* of the operator, and an ordered list of variables called *arguments*. Each variable is possibly annotated with a *binding pattern* [25] that indicates that the argument must be *bound* in order for the query to be executed (denoted with the annotation `$`). Variables with no annotation are *free*.

2. A *body* which is a conjunction of atomic formulae whose predicate symbols denote relations in the world model.[3]

For clarity, we depict operators as expressions with the head on the left, an implication symbol, and the body on the right:[4]

$$op(\mathtt{X}_1,\ldots,\mathtt{X}_n) \Rightarrow \mathtt{rel}_1(\ldots,\mathtt{X}_i,\ldots) \wedge\ldots\wedge \mathtt{rel}_m(\ldots,\mathtt{X}_j,\ldots)$$

This specification says that when *op* is executed it will return some number of tuples of data, where each tuple may be thought of as an assignment of values to the head's arguments $\mathtt{X}_1,\ldots,\mathtt{X}_n$. The operator specification dictates that for each tuple returned, the logical formula formed by replacing all occurences of the arguments in the body with the corresponding tuple values is satisfiable.

For example, one can model the UNIX finger command with the following operator:

$$finger(\mathtt{F,L,\$E,O,Ph}) \Rightarrow \mathtt{email(F,L,E)}\wedge\mathtt{office(F,L,O)}\wedge\mathtt{phone(O,Ph)}$$

Intuitively, this means that when given an email address (the bound[5] variable `$E`), *finger* produces a set of variable bindings for the free variables `F` `L`, `O` and `Ph`. For example, when `E` is bound to `"sam@cs"`, the following tuples might be returned:

⟨`"Sam","Smith","sam@cs","501","542-8907"`⟩
⟨`"Sam","Smith","sam@cs","501","542-8908"`⟩

The relation `office(F,L,O)` appears in the body of *finger* , hence we can conclude that `office("Sam","Smith","501")` is true, and we know that office `"501"` has at least two phones: `"542-8907"` and `"542-8908"`.

[3]The body can also contain numerical constraints (*i.e.*, built-in predicates [27, p101]), and Occam can generate appropriately constrained plans, but we do not discuss this aspect of Occam in this paper.

[4]A brief comment on notation: we follow Prolog conventions, hence symbols beginning with a capital leter denote variables. All free variables are universally quantified. All variables in the body that don't appear in the arguments are said to be *unbound* and are considered existentially quantified inside the scope of the free variables. In general, we use `typewriter font` for relations in the world model and *italics* for operators and queries. We use the function Args($O$) to denote the arguments of the operator $O$; Body($O$) denotes the operator's body, and Name($O$) denotes its name.

[5]Since the binding annotations are required only for variables appearing as arguments, the use of `$E` in the head denotes the same variable as `E` in the body.

It is important to note that operators are not guarenteed to return *all* tuples that are conceptually part of the world model relations. This is the appropriate semantics for operators, since most data sources are incomplete — telephone white pages don't show *all* phone numbers, since some are unlisted. The SABRE flight database doesn't record *all* flights between two points on a given day, because some airlines are too small to be included. As a result of this inherent database incompleteness, one must often execute multiple operators in order to be sure that one has retrieved as many tuples as possible.[6]

As another concrete example, the operator *userid-room* generates the email addresses `E` for the occupants in office `O`.

*userid-room*(`$O,E`) $\Rightarrow$ `office(F,L,O)`$\wedge$ `email(F,L,E)`

Note that this operator does not return values for the first and last names associated with each email address `E`. Nevertheless, variables (*e.g.*, `F` and `L`) ranging over these attributes of `email` and `office` are necessary to define the query in terms of the relations in the world model; such variables are said to be *unbound*. The interpretation is as follows: if *userid-room* returns a tuple such as $\langle$`"501","sam@cs"`$\rangle$ then

$\exists$`F,L` such that `office(F,L,"501")`$\wedge$ `email(F,L,"sam@cs")`

Our examples of *finger* and *userid-room* illustrate encodings of legacy systems. For example, UNIX *finger* may be thought of as having access to relational data about names, email addresses, phone numbers and offices, but it does not support arbitrary relational operations. If one wished to know the email address of everyone whose phone number was `"555-1212"` *finger* would be of little use. Binding patterns are a convenient way to describe legacy information sources, because they indicate the type of queries supported by that site. When a system supports several types of query (but doesn't support full relational operations) it can be described with several operators. Full relational databases are simply described using operators with no bound variables.

Although our syntax for operators looks very different from traditional STRIPS [11] or ADL [23] planning operators, there are many similarities. In

[6] If one knows that a site *does* contain all tuples, then one could specify this by using $\Leftrightarrow$ to separate the operator head and body. Given such a specification, one could perform local closed world reasoning [7, 12, 18] to eliminate operators from consideration, but we do not discuss the matter in this paper.

particular, while Occam operators have no causal preconditions, the bound arguments in an operator's head represent a form of knowledge precondition [21, 22] that is equivalent to the `findout` goals of UWL [8]. There are no causal effects, but the body of an operator is similar to a UWL `observe` effect. The similarity is not exact, however, because execution of an Occam operator may generate an unbounded number of values.

## 2.2 Representing an Information Gathering Query

Queries are very similar to operators: they also have heads and conjunctive bodies, but the direction of implication is reversed. The interpretation is that any tuple satisfying the body (a conjunction of world relations) satisfies the query. For example, if we want to know the first-names of the occupants in an office, we can issue the query

*query-for-first-names*(`$O, F`) $\Leftarrow$ `office(F,L,O)`

Note that this query has two arguments, `O` and `F`; the binding pattern indicates that `O` must be bound (*e.g.*, to `"429"`) before the query is executed. Since `F` has no `$` annotatation, the query is requesting a set of values for that variable. For example, if Joe Researcher and Jane Goodhacker are the occupants of office 429, then the tuples ⟨`"429","Joe"`⟩ and ⟨`"429","Jane"`⟩ are possible answers for this query.

# 3 Plans & Solutions

If some site stored the complete `office` relation in a relational database, then *query-for-first-names*(`"429",F`)[7] could be answered by evaluating a simple SQL expression. However, if the data repository doesn't support relational operations or if the data forming the `office` relation is distributed across multiple sites, then the situaion is more complex.

For example, if one is limited to the operators described above, then the best way to satisfy the example query is to first execute *userid-room*, which returns bindings for the email addresses of the office's occupants. Next one would execute *finger* repeatedly for each binding of `E` and discard all information returned except for the first-name.

[7] For clarity, we often substitute a constant in place of a bound variable when writing queries.

## 3.1 Plans

Formally a *plan* has the same representation as an operator whose body is an ordered conjunction of operator instances.[8] For example, the previous section's example can be encoded as the two step plan:

$$\mathit{plan}(\texttt{"429"},\texttt{F}) \Rightarrow \mathit{userid\text{-}room}(\texttt{"429"},\texttt{E}) \wedge \mathit{finger}(\texttt{F},\texttt{L},\texttt{E},\texttt{"429"},\texttt{Ph})$$

There are two ways of interpreting the body of a plan and both are important. On the one hand, the body can be viewed as a logical conjunction in which case the order is unimportant. On the other hand, the body can be viewed procedurally in which case the order is very important; in particular, the order lets one determine if operator binding patterns are satisfied.

A plan's head specifies what information is actually returned to the user. For example, although execution of *finger* gathers information about people's last names, the plan shown doesn't return this information to the user.

## 3.2 Solutions to a Query

We say a plan $\mathit{plan}(\texttt{X}_1,\ldots,\texttt{X}_n) \Rightarrow O_1 \wedge \ldots \wedge O_k$ is a *solution* to the query $\mathit{query}(\texttt{Y}_1,\ldots,\texttt{Y}_n) \Leftarrow \texttt{rel}_1(\ldots,\texttt{Y}_i,\ldots) \wedge \ldots \wedge \texttt{rel}_m(\ldots,\texttt{Y}_j,\ldots)$ if the following two criteria are satisfied:

1. The binding patterns of the plan's operator instances are satisfied. Specifically, if `$V` is a bound argument of $O_j$ then `V` must be used as a free argument to some other operator instance $O_i$ where $i < j$ or else a value for `V` must be a bound argument in the query head [25].

2. All tuples satisfying $\mathit{plan}(\texttt{X}_1,\ldots,\texttt{X}_n)$ must satisfy $\mathit{query}(\texttt{X}_1,\ldots,\texttt{X}_n)$. In other words, the following implication must hold:

$$\forall \texttt{c}_1,\ldots,\texttt{c}_n \;\; \mathit{plan}(\texttt{c}_1,\ldots,\texttt{c}_n) \Rightarrow \mathit{query}(\texttt{c}_1,\ldots,\texttt{c}_n)$$

   where each $c_i$ is a constant.

[8] In contrast to an operator, which is an abstract representation of an information source's capabilities, an *operator instance* represents the action of *accessing* a specific information source. Formally, an operator instance is an atomic formula whose predicate symbol is an operator name. Thus $\mathit{userid\text{-}room}(\texttt{O}_1,\texttt{E}_1)$, and $\mathit{userid\text{-}room}(\texttt{O}_2,\texttt{E}_2)$ are distinct operator instances.

For example, the plan *plan* in section 3.1 is a solution to *query-for-first-names* because

1. The binding patterns are satisfied: *userid-room* has only one bound variable, `$0`, and it is bound to `"429"` by the query, and since execution of *userid-room* binds `E`, the binding template of *finger* is satisfied as well.

2. Every tuple returned by the plan satisfies *query-for-first-names*(`"429"`, `F`). To see that this is true, suppose the tuple of constants $\langle c_1, c_2 \rangle$ is returned by the plan. The following logical implications hold:

$$\begin{aligned} \textit{plan}(c_1,\ c_2) &\Rightarrow \textit{userid-room}(c_1, \mathtt{E}) \wedge \textit{finger}(c_2, \mathtt{L}, \mathtt{E}, c_1, \mathtt{Ph}) \\ &\Rightarrow \mathtt{office}(\mathtt{F}_0, \mathtt{L}_0, c_1) \wedge \mathtt{email}(\mathtt{F}_0, \mathtt{L}_0, \mathtt{E}) \wedge \mathtt{email}(c_1, \mathtt{L}, \mathtt{E}) \wedge \\ &\quad \mathtt{office}(c_2, \mathtt{L}, c_1) \wedge \mathtt{phone}(\mathtt{0}, \mathtt{Ph}) \\ &\Rightarrow \mathtt{office}(c_2, \mathtt{L}, c_1) \\ &\Rightarrow \textit{query-for-first-names}(c_1, c_2) \end{aligned}$$

   Thus any tuple $\langle c_1, c_2 \rangle$ satisfying *plan*($c_1$, $c_2$) also satisfies *query-for-first-names*($c_1, c_2$), hence the plan is a solution to the query.

# 4 Planning to Gather Information

Figure 1 presents the Occam forward-chaining planner. As input, Occam takes a query and set of operators. As output, Occam produces a set of plans, each of which is guaranteed to be a solution. Starting from the empy sequence, Occam searches the space of totally ordered sequences of operator instances (*i.e.* plan *bodies*). Since there is no bound on the length of useful plans (appendix A), Occam's search proceeds until all alternatives have been exhausted, or a resource bound is exceeded. At each stage a sequence of operator instances, *Seq*, is removed from *Fringe* and is expanded by postpending an instance of each potential operator. Since operators can be instantiated in several ways (figure 2), expanding *Seq* will typically cause many new sequences to be added to *Fringe*. FindSolutions determines if any of these sequences can be elaborated into a solution plan; this is akin to evaluating the modal truth criterion [4] and is explained in section 4.2; Occam adds all newly discovered solutions to *Sol*, but in any case *every* sequence is kept on *Fringe* because its children might lead to qualitatively different solutions.

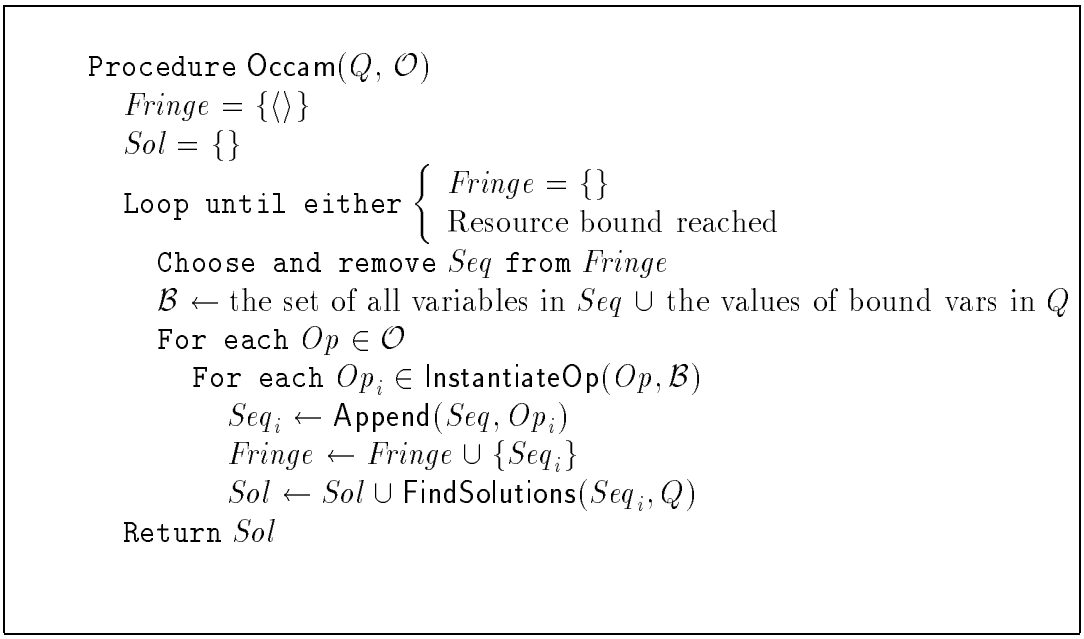


Figure 1: A forward-chaining algorithm for generating query plans; input $Q$ is a query, and $\mathcal{O}$ is a set of operators; the output is a set of solutions.

## 4.1 The Example, Revisited

Suppose Occam is called on the *query-for-first-names* example. When the empty sequence is removed from *Fringe*, Occam considers adding instances of operators *finger* and *userid-room*. Since there are no instances in the empty sequence, $\mathcal{B}$ is assigned the value {"429"} because that is the only constant provided as input by the query.

When InstantiateOp is called with *userid-room*, the procedure must create *Val* sets corresponding to *userid-room*'s two arguments, the bound O and the free E. Potentially, O could be assigned any value (there is only one) in $\mathcal{B}$ that has type which is consistent with offices. Since both "429" and O are of type office, *Val*(O) = {"429"}; if there had been a type conflict, then *Val*(O) would have been empty and InstantiateOp would have returned no instances. Since E is free, *Val*(E) is assigned a set containing a newly generated variable, $\{E_0\}$.

Since both *Val* sets are singletons, there is only one pair in the cross product. Hence, InstantiateOp returns a single instance to Occam: *userid-room*("429", $E_0$).

In some later iteration of Occam, $Seq$ = *userid-room*("429", $E_0$) will be

```
Procedure InstantiateOp(Op, B)
  Instances ← {}
  For each variable, V_i, in Args(Op)
    If V_i is bound, then Val(V_i) ← {X ∈ B | SameType(X, V_i)}
    Else if V_i is free, then Val(V_i) ← {a newly generated variable}
  For each tuple ⟨X_1, ..., X_n⟩ in the cross product Val(V_1) × ... × Val(V_n)
    Generate a new operator instance Op_i such that
      Name(Op_i) ← Name(Op)
      Args(Op_i) ← ⟨X_1, ..., X_n⟩
    Instances ← Instances ∪ {Op_i}
  Return Instances
```

Figure 2: Instantiating an operator; input $Op$ is an operator and $\mathcal{B}$ is the set of bound variables, output *Instances* is a set of operator instances.

removed from *Fringe*. $\mathcal{B}$ will now be assigned the value, {"429", $\mathtt{E}_0$}. This is Occam's way of noting that after executing *userid-room*, Occam will have a set of possible values for $\mathtt{E}_0$ and thus can use that variable when instantiating future instances that have bound arguments.

Once again Occam will consider adding instances of *finger* and *userid-room*. When it chooses the former, it must create *Val* sets for *finger*'s arguments: F, L, E, O, Ph. Since all of these arguments except E are free, their *Val* sets will contain a single newly generated variable each, *e.g.* $\{\mathtt{F}_1\}, \{\mathtt{L}_1\}, \{\mathtt{O}_1\}, \{\mathtt{Ph}_1\}$. Although there are two members of $\mathcal{B}$, only one has type email address, so $Val(\mathtt{E}) = \{\mathtt{E}_0\}$. Therefore InstantiateOp returns a single instance to Occam: $finger(\mathtt{F}_1, \mathtt{L}_1, \mathtt{E}_0, \mathtt{O}_1, \mathtt{Ph}_1)$, and we have a sequence

$$userid\text{-}room(\text{"429"}, \mathtt{E}_0) \wedge finger(\mathtt{F}_1, \mathtt{L}_1, \mathtt{E}_0, \mathtt{O}_1, \mathtt{Ph}_1)$$

This sequence is added to *Fringe*, and in addition it is passed to FindSolutions, in order to see if it could be the basis for a solution to the query.

## 4.2 Finding Solutions from Sequences

The previous section described how Occam enumerates the space of totally ordered sequences of operator instances. This section explains how the

FindSolutions function tests each sequence to see if it encodes one or more solutions to the query. Note, first, that there is a difference between a *plan* and a *sequence* of operator instances. A plan is represented as an operator, and as such it has both a head and a body; the body determines which actions get executed while the head determines what data gets returned.

When given a sequence, $O_1 \wedge \ldots \wedge O_k$, of operator instances, FindSolutions determines whether there exist any plans of the form $plan(\mathtt{X}_1, \ldots, \mathtt{X}_n) \Rightarrow O_1 \wedge \ldots \wedge O_k$ that are solutions to the query. This test is somewhat akin to the Modal Truth Criterion [4] which tests a partially ordered (hence incomplely specified) plan to see if any solution exists. In the case of Occam, a totally ordered sequence of operators is underspecified because there could be several (or no) heads which render it a solution.

Recall that section 3.2 defined the two requirements for a plan to be a solution to a query. First, the binding patterns of the plan body's operator instances must be satisfied. FindSolutions doesn't need to check this criterion because InstantiateOp is careful in its choice of *Val* sets so that every bound variable is only instantiated with acceptable values. The second condition was that all tuples satisfying $plan(\mathtt{X_1}, \ldots, \mathtt{X_n})$ must satisfy $query(\mathtt{X_1}, \ldots, \mathtt{X_n})$. As shown in Figure 3, the FindSolutions function takes a sequence and generates the set of all plans (having the sequence for their body) whose tuples are guaranteed to satisfy the query. These plans are thus solutions.

Underlying the operation of FindSolutions is the notion of a *containment mapping* between two horn clauses [28, p881]. A containment mapping from query $Q$ to the formula $E$ is a function $\tau$ mapping symbols in $Q$ to symbols in $E$. If there exists a mapping such that the $\tau(\mathsf{Body}(Q))$ is a subset of the body of $E$ while $\tau(\mathsf{Args}(Q))$ equals the arguments of $E$, then $E$ logically entails $Q$.[9]

When FindSolutions is given a sequence of operator instances, *i.e.* a potential *plan body*, it first computes the *expansion* of the sequence by setting $E$ to the conjunction of the *bodies of the operators in the sequence.* $\mathcal{V}_E$ and $\mathcal{V}_Q$ are defined so that FindSolutions can enumerate the space of potential containment mappings.[10] If it can find a containment mapping from the query to the

[9]Containment mapping suffices for logical entailment in our language. However, for a more expressive description language (such as that described in [18]) containment mapping must be augmented with more powerful tools such as subsumption algorithms and inequality reasoning. In fact, the Occam implementation handles mathematical constraints, but we do not discuss them in this paper.

[10]In practice, the use of type information and other optimizations allows a much more ef-

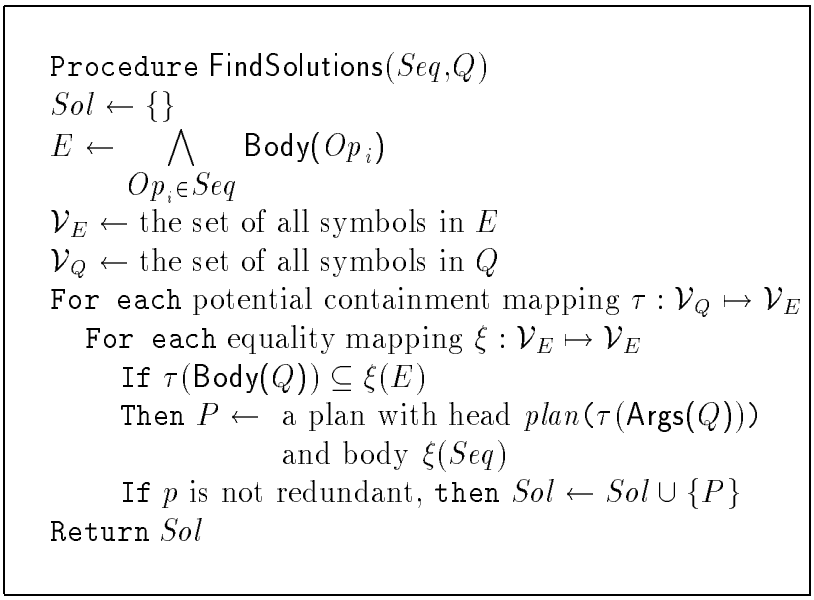


Figure 3: Finding solutions; the input is a query and a sequence of operator instances; the output is a set of plans (the solutions).

expansion $E$, then this enables the construction of a plan head guaranteeing that all tuples returned by the plan will satisfy the query. FindSolutions also considers possible *equality mappings* which have the effect of requiring that two or more variables in $E$ are constrained to be equal.

## 4.3 The Example, Continued

From section 4.1, recall that Occam had just generated the promising sequence *userid-room*("429", $\mathtt{E_0}$) $\wedge$ *finger*($\mathtt{F_1}, \mathtt{L_1}, \mathtt{E_0}, \mathtt{O_1}, \mathtt{Ph_1}$), and asked FindSolutions to see if any plans could be made (with this conjunction for their body) in order to solve *query-for-first-names*. Given these arguments, FindSolutions expands the sequence, giving $E$ the following value:

$\mathtt{office}(\mathtt{F_0}, \mathtt{L_0}, \mathtt{"429"}) \wedge \mathtt{email}(\mathtt{F_0}, \mathtt{L_0}, \mathtt{E_0}) \wedge$
$\mathtt{email}(\mathtt{F_1}, \mathtt{L_1}, \mathtt{E_0}) \wedge \mathtt{office}(\mathtt{F_1}, \mathtt{L_1}, \mathtt{O_1}) \wedge \mathtt{phone}(\mathtt{O_1}, \mathtt{Ph_1})$

$\mathcal{V}_Q$ becomes {"429", F, L} and $\mathcal{V}_E$ becomes $\{\mathtt{F_0}, \mathtt{L_0}, \mathtt{"429"}, \mathtt{E_0}, \mathtt{F_1}, \mathtt{L_1}, \mathtt{O_1}, \mathtt{Ph_1}\}$.

ficient algorithm than this brute-force enumeration of containment and equality mappings. See the long version of the paper.

Next, FindSolutions tries different ways to map variables from $\mathcal{V}_Q$ to $\mathcal{V}_E$. Eventually, it considers the following mapping: $\tau(\texttt{"429"}) = O_1$, $\tau(F) = F_1$, and $\tau(L) = L_1$. Suppose $\xi$ is the identity mapping. Applying $\tau$ to the query body yields the singleton sequence $\texttt{office}(\texttt{F}_1, \texttt{L}_1, \texttt{O}_1)$, which matches one of the conjuncts in $E$. Therefore we make a new plan $P$:

$$\textit{plan}(O_1, F_1) \;\Rightarrow\; \textit{userid-room}(\texttt{"429"}, \texttt{E}_0) \;\wedge\; \textit{finger}(\texttt{F}_1, \texttt{L}_1, \texttt{E}_0, \texttt{O}_1, \texttt{Ph}_1)$$

Since $P$ is not redundant (defined in section 4.5) it is saved as a solution in *Sol*. In this example, there are no other solution plans with *userid-room* $\wedge$*finger* as body, but in some cases there exist several heads that make a sequence into a solution. When this happens, FindSolutions returns all such plans.

## 4.4 Transformations Based on Equality Mappings

FindSolutions's inner loop enumerates the space of equality mappings, functions of the form $\xi : \mathcal{V}_E \mapsto \mathcal{V}_E$. By performing this search, FindSolutions considers the possibility of constraining one or more of the variables in the expansion to be equal. Although equality mappings weren't important in the previous example, sometimes they are necessary in oder to recognize a solution. For example consider the query

$$\textit{query-same-names}(\texttt{\$E, F}) \Leftarrow \texttt{email(F,F,E)}$$

which looks for people with a given email address whose firstnames and lastnames are the same. The sequence

$$\textit{finger}(\texttt{F}_0, \texttt{L}_0, \texttt{\$E}, \texttt{O}_0, \texttt{Ph}_0)$$

which has expansion

$$\texttt{email}(\texttt{F}_0, \texttt{L}_0, \texttt{E}) \wedge \texttt{office}(\texttt{F}_0, \texttt{L}_0, \texttt{O}_0) \wedge \texttt{phone}(\texttt{O}_0, \texttt{Ph}_0)$$

might lead to a solution, but note that there is no possible containment mapping from $\texttt{email}(\texttt{F}, \texttt{F}, \texttt{E})$ to $\texttt{email}(\texttt{F}_0, \texttt{L}_0, \texttt{E})$ because the image of F would have to be both $\texttt{F}_0$ *and* $\texttt{L}_0$. FindSolutions generates solution by constraining $\texttt{L}_0 = \texttt{F}_0$) with the equality mapping that sets $\xi(\texttt{F}_0) = \texttt{L}_0$. As a result $\xi(E)$ contains the conjunct $\texttt{email}(\texttt{F}_0, \texttt{F}_0, \texttt{E})$, and the containment mapping $\tau(\texttt{F}) = \texttt{F}_0$, $\tau(\texttt{E}) = \texttt{E}$ demonstrates logical entailment. Thus FindSolutions recognizes solution corresponding to executing a *finger* command and then discarding all tuples whose first and last names are not equal.

## 4.5 Redundant Solutions

We call a solution *redundant* if we can eliminate operator instances from the plan and still obtain a solution. The last line of FindSolutions checks to see if a plan is redundant before adding it to the set of solutions to be returned. To see why this check is absolutely essential, note that if a sequence of operator instances corresponds to a solution then *every supersequence* will also generate that solution. Furthermore, recall that Occam keeps all sequences on the *Fringe*, even when they have produced solutions. Thus it is crucial to discard redundant solutions.

One might ask *why* Occam keeps solution sequences on the *Fringe*. The answer is that some supersequences yield qualitatively different solutions that should not be ignored. To understand these points, consider the following artificial set of operators:

$op_1$(X) $\Rightarrow$ $\texttt{rel}_1$(X)
$op_2$($X,Y) $\Rightarrow$ $\texttt{rel}_2$(X,Y)
$op_3$($X,Y) $\Rightarrow$ $\texttt{rel}_2$(X,Y) $\wedge$ $\texttt{rel}_1$(Y)

Now given the query *query*(X) $\Rightarrow$ $\texttt{rel}_1$(X), consider the following plans:

*plan1*(A) $\Rightarrow$ $op_1$(A)
*plan2*(A) $\Rightarrow$ $op_1$(A)$\wedge op_2$(A,B)
*plan3a*(A) $\Rightarrow$ $op_1$(A)$\wedge op_3$(A,B)
*plan3b*(B) $\Rightarrow$ $op_1$(A)$\wedge op_3$(A,B)

Note that *plan1* is a solution to query. Although *plan2* is also a solution, it is redundant because elimination of the second conjunct yields *plan1* which is a solution. Because the sequence $op_1$(A)$\wedge op_3$(A,B) is a superset of *plan1*'s body, it also leads redundant solution *plan3a*. But this same sequence also leads to a *non-redundant* solution *plan3b*. Since this latter plan might return a different set of tuples, than *plan1*, it is noticed and returned by FindSolutions.

Space considerations preclude details, but we assert that checking for redundancy is very fast, taking time that is $O(l^2eg)$ where $l$ is the length of the plan, $e$ is the number of relations in the expansion and $g$ is the number of relations in query body.

## 4.6 Formal Properties

Space limitations preclude actual proofs of soundness and completeness, but we give an intuitive argument. Soundness follows from two facts: 1)

InstantiateOp only returns operator instances whose bound variables are chose from the set $\mathcal{B}$ which Occam calculates to be precisely those satisfying the binding constraints, 2) FindSolutions constructs a containment mapping which suffices to compute logical entailment for this language [28].

Completeness follows from the fact that Occam performs an exhaustive search (to a depth bound) of all possible sequences of operator instances that satisfy the binding constraints. Since FindSolutions enumerates all possible heads, all possible plans are considered, and all non-redundant plans are returned.

As it stands, the algorithm is also reasonably efficient. But the next section describes two optimizations that improve performance considerably.

# 5 Reducing Search

In this section we describe two optimizations (one obvious, the other less so) that reduce the number of plan bodies explored by Occam's search process. Duplicated operator instance pruning (section 5.1) eliminates redundant instances. Shuffled sequence pruning (section 5.2) achieves the efficiency benefits of a partial-order representation without the attendant complexity. Both of these techniques are domain-independent and completeness-preserving—they do not cause the planner to miss useful solutions. Furthermore, as we demonstrate with experiments below, the techniques are complementary; when used in combination they provide on average an order of magnitude speedup.

## 5.1 Pruning Plans with Duplicate Operator Instances

We say that two instances, $O$ and $O'$, of an operator are *equivalent* (written $O \stackrel{op}{=} O'$) if all the bound arguments of $O$ are equal to the variables in $O'$. For example *userid-room*(A,B) $\stackrel{op}{\neq}$ *userid-room*(C,B), but *userid-room*(A,B) $\stackrel{op}{=}$ *userid-room*(A,C). We reject any sequence that contain two equivalent operator instances. This test is best accomplished by incrementally checking for the presence of an equivalent instance before postpending new instances returned by the call to InstantiateOp. That is, we reject the sequence if the new operator instance has the same bound arguments as an existing step, since executing the same operator with same bound arguments twice will not get us any new tuples. For example, the plan

*userid-room*("429",E) $\wedge$ *finger*(F,L,E,O,Ph) $\wedge$ *userid-room*("429",$E_1$) should be rejected because *userid-room* is executed twice with the same bound argument "429".

## 5.2 Pruning Shuffled Sequences

Since Occam generates sequences that are totally ordered, it frequently considers different ordering of operator instantances when the precise order does not matter at all. For example, given the following operators

$op_1$(X,Y) $\Rightarrow$ $\mathtt{rel}_1$(X,Y)

$op_2$($X,Y) $\Rightarrow$ $\mathtt{rel}_2$(X,Y)

the following two incomplete sequences

$s_1$: $op_1$(A,B) $\wedge op_2$(A,C) $\wedge$ $op_2$(B,D)

$s_2$: $op_1$(A,B) $\wedge op_2$(B,D) $\wedge$ $op_2$(A,C)

would be considered by Occam, but in fact we can discard one or the other without losing completeness.

One way to reduce this combinatorial explosion is to use a partial-order respresentation [3, 20], but we adopt a simpler approach: enforcing a canonical ordering which eliminates redundant permutations.

We say operator instance $O_i$ is *dependent* on $O_j$ if either

- $O_i$ has a bound argument that appears as a free variable in $O_j$, or
- There exists an instance $O_k$ such that $O_i$ is *dependent* on $O_k$ and $O_k$ is *dependent* on $O_j$.

We say that two instances are *independent* if neither is dependent on the other. If two operator instances are independent, then Occam does not need to consider both ordering permutations.

To avoid this redundancy, we assign an (arbitrary) unique number InstanceID($O$) to each operator instance $O$; InstanceID($O_i$) = InstanceID($O_j$) if and only if $O_i \stackrel{op}{=} O_j$. When creating new sequences by adding operator instances to an existing sequence, Occam prunes the creation if the new instance $O$ is independent of an existing operator instance $O_i$ and InstanceID($O$) $<$ InstanceID($O_i$).

Returning to the example above, suppose we assign InstanceID($op_1$(A,B)) = 1, InstanceID($op_2$(A,C)) = 2 and InstanceID($op_3$(B,D)) = 3. To construct $s_2$ we add $op_2$(A,C) to the sequence $op_1$(A,B) $\wedge op_2$(B,D). However $op_2$(B,D) is independent of $op_2$(A,C), and InstanceID($op_2$(A,C)) $<$ InstanceID($op_2$(B,D)), so this addition is redundant and $s_2$ should be pruned. It should be obvious

that $s_1$ is the only sequence that consists of all 3 of the operator instances in the search space.[11]

### 5.3 Experimental Validation

Our experiments consist of 5 problems in 4 domains. A precise description of the domains is precluded by space limitations, but we provide the following brief summary:

- The Parent domain (defined in [25]) has two operators.
- The Patho domain consists of the three operators from section A.
- The Car domain models relational databases containing price information on foreign and domestic cars with five operators that use numerical constraints to encode price and date restrictions.
- The People domain (derived from [9]) encodes Internet white page operations with ten operators.

In each experiment Occam exhaustively explores all sequences up to a certain length; the number of sequences explored and the time taken for each experiment is shown in Table 1. Each experiment is run with Occam (the "plain" column), Occam with duplicated operator instance pruning (the "no duplicate" column) and Occam with duplicated operator instance pruning and shuffled sequence pruning (the "no shuffle" column). This preliminary experiment shows that our search control optimizations provide two orders of magnitude speedup.

## 6 Finding Simplest Plans

Network information services are potentially slow and expensive; therefore one important goal of Occam is to find the simplest plans that gather information requested by the user. Occam achieves this goal by recording the price and expected time required by the information source requires. Occam uses this information to guide its search by picking the expected *least cost*

[11] The sequence generated by this ordering technique may not be optimal for execution, but Occam may reorder operator instances during runtime to achieve execution efficiency.

| Query | | plain | no duplicate | no duplicate no shuffle |
|---|---|---|---|---|
| find-grandparent (depth 7) | explored | 46232 | 8852 | 413 |
| | time | 23 | 3 | $< 1$ |
| Patho q(X) (depth 7) | explored | 598443 | 163531 | 10024 |
| | time | 364 | 65 | 2 |
| Car query (depth 7) | explored | 97655 | 97655 | 2310 |
| | time | 975 | 975 | 8 |
| query-for-first-names (depth 6) | explored | 62808 | 22307 | 8480 |
| | time | 346 | 88 | 9 |
| find-email (depth 4) | explored | 14249 | 11141 | 5257 |
| | time | 19 | 16 | 2 |

Table 1: Performance of Occam; *time* is in CPU seconds (on a Silicon Graphics Indy under Allegro Common Lisp 4.2) , and *explored* refers to number of sequences visited in the search space. *Depth* refers to the maximum length of sequences considered.

sequence from *Fringe* (figure 1). Users can also impose *resource bounds* on a query (*e.g.*, specifying a price bound of zero on a query prevents Occam from considering any sequence that cost money). Occam finds the simplest plans using these criteria.

However, estimating the cost of a sequence is not as straightforward as it might appear. In particular, operators encoding legacy systems may need to execute multiple times in a sequence, once for each tuple returned from another information source. Occam copes with this problem by estimating the *execution complexity* of a sequence. Occam can estimate the number of tuples the information source returns on average, and then multiply this number with the cost of the operators that use tuples from this source to arrive at an estimate of the total execution cost.

# 7 Related Work

Several researchers in database community are concerned with the integration of heterogenous databases. Prominent projects include the Information Manifold [14] and the Tsimmis project [5]. From Tsimmis, we adopt the no-

tion of notion of binding templates [25]. However, for the most part, Tsimmis assumes information integration is done manually, while our work focuses on automating the information-integration process. In particular, Tsimmis uses binding templates to model information sources and generate data abstractions called wrappers [13], whereas in Occam they are used to constrain search during the automatic construction of valid plans.

The rest of our representation language is based on the encodings described in [18], but in contrast to this work we provide implemented algorithms for generating query plans when site descriptions include binding annotations. In addition, we describe several optimizations and demonstrate their effectiveness experimentally. On the other hand, the description language in [18] provides a more expressive type hierarchy than that used by Occam.

The bulk of prior work on AI planning systems [1] assumes that execution of an operator instance has a causal effect on the world. In contrast, our approach eschews updates to the information sources and instead focuses solely on information gathering. This allows a much simpler and faster planning algorithm, since Occam need not worry about sibling-subgoal interactions or threats to causal links. Although some researchers have argued that partial-order planners are more efficient than total-order planners [19, 20, 3], their arguments do not apply to the information gathering problem. Occam's shuffled-sequence pruning technique (section 5.2) provides all the benefits of a partial-order representation with reduced complexity.

Several planning systems were designed specifically for information gathering. For example, the XII planner [12] guides the Internet Softbot [9], and the Sage planner [15] controls the SIMS information system [2]. Like Occam, both XII and Sage specify transformations between the information produced by a remote site and an internal world model. But Occam allows a more general class of transformations in several ways. Neither Sage nor XII can represent information sources that generate information which translates into partially specified sentences in the world model. For example, both these systems are unable to model *userid-room* in terms of the `office` and `email` relation schemata, because they can't handle Occam-style unbound variables. Furthermore, Sage and XII are unable to represent an incomplete source that returns variable number of tuples. On the other hand, both Sage and XII can handle goals with negation and disjunction, In addition, XII performs sophis-

ticated local closed world reasoning.[12] Finally, Sage allows parallel execution in its control of multiple database operations.

Both Sage and XII interleave planning and execution, but another approach is the generation of conditional plans [17, 24, 6]. Most of the planners described above have significant combinatorial explosions and require domain-specific, search control for anything but small problems. For example, XII requires considerable control knowledge in order to handle problems that appear comparable to those in our People domain. A major contribution of our work is the development of a domain-independent, sound and complete algorithm that runs at practical speeds.

Finally, the language we use in Occam is quite compact. For example, unlike XII's representation, quantification is implicit and hence concise. Binding patterns allow a parsimonious encoding of knowledge preconditions. The world model is more flexible, since relations can be combined using unbound variables and implicit joins. As a result, Occam domain theories are considerably shorter (*e.g.*, less than a third the size) thus easier to define and debug.

# 8 Conclusion

We have described a novel planning algorithm, Occam, that is optimized for the problem of gathering and integrating Internet information sources. Since most sites on the Internet do not allow updates, our action language does not support the notion of causal change. Because most information sources are accessible by simple actions such as opening an HTTP connection, our language does not support the notion of traditional preconditions. These dramatic simplifications allow a much simpler planning algorithm. Since Occam need not consider sibling-subgoal interactions, it requires no representation of causal link, and it need not perform promotion nor demotion. Indeed, section 5.2 shows Occam has no need for a partial-order representation.

Simplifying the action language in some respects allowed us the opportunity to increase it's expressiveness in other ways. While Occam actions don't have traditional preconditions, they may have *knowledge preconditions*. By reasoning about the capabilities of different information sources, Occam can

[12] XII is schizophrenic in the sense that it can reason about when *it* has complete information and when it does not, but it is incapable of representing when a *remote information source* has complete information and when it does not.

extract data from both legacy systems and full relational databases. Although Occam need not represent changes to the world state, it does reason about changes to the *state of information* during the course of the plan. Unlike previous implemented systems, Occam can reason about the fact that information sources may contain an unbounded amount of information, without assuming that the source contains all possible information. Because of this, Occam handles partial goal satisfaction: when no single plan can gather all information, Occam generates alternatives that may be executed in parallel to collect as much information as possible.

We argued that Occam is both sound and complete. In addition, Occam is efficient as we demonstrated in preliminary empirical tests. We described improvements that speed planning even more. In particular, shuffled-sequence pruning (along with Occam's ability to reason about the type of relational attributes) is very effective.

In the future we hope to incorporate local closed world information [7, 18] into our planner so that Occam can reason about situations when it has exhausted all information gathering alternatives. We are also integrating speed and cost metrics into the site descriptions so that Occam can answer queries in a manner that optimizes a user-specific utility function. Finally, we are invesgating alternative search paradigms such as backward-chaining which may create a smaller plan space than our current forward-chaining algorithm.

# A Bounds on Plan Length

Recall that for a plan to be a solution, every tuple satisfying *plan* must logically entail *query* (section 3.2). Using a stronger definition of solution (where the entailment is bidirectional) Rajaraman *et al.* [25] have proven that one need not consider plans of unbounded length when searching for tight solutions. Specifically, they showed that for a query with $n$ conjuncts and $m$ unique variables, if a "bidirectional" solution exists, then there is an equivalent solution with at most $m + n$ conjuncts.

Unfortunately, this bound does not apply to our definition of solutions. To see this, consider the artifical set of three operators:

$op_1$(X) $\Rightarrow$ $\texttt{rel}_1$(X)
$op_2$($X,Y) $\Rightarrow$ $\texttt{rel}_2$(X,Y)
$op_3$($X) $\Rightarrow$ $\texttt{rel}_3$(X)

and suppose that the following query has been issued:

$q(\mathtt{X}) \Rightarrow \mathtt{rel}_3(\mathtt{X})$

Note that $q(\mathtt{Z}) \Rightarrow op_1(\mathtt{Z}) \wedge op_3(\mathtt{Z})$ is a solution (but not tight); it returns elements satisfying both $\mathtt{rel}_3$ and $\mathtt{rel}_1$, hence may miss elements that satisfy $\mathtt{rel}_3$ but not $\mathtt{rel}_1$. Note further that the following plan is also a solution:

$$q(\mathtt{Z}) \Rightarrow op_1(\mathtt{A}) \wedge op_2(\mathtt{A},\mathtt{B}) \wedge op_2(\mathtt{B},\mathtt{C}) \wedge \ldots \wedge op_2(\mathtt{Y},\mathtt{Z}) \wedge op_3(\mathtt{Z})$$

This plan expands to

$$q(\mathtt{Z}) \Rightarrow \mathtt{rel}_1(\mathtt{A}) \wedge \mathtt{rel}_2(\mathtt{A},\mathtt{B}) \wedge \mathtt{rel}_2(\mathtt{B},\mathtt{C}) \wedge \ldots \wedge \mathtt{rel}_2(\mathtt{Y},\mathtt{Z}) \wedge \mathtt{rel}_3(\mathtt{Z})$$

which is contained in the body of $q$, but which is *not* contained in any shorter plan. Hence this longer plan may return tuples that are missed by shorter plans. Unfortunately, there are an infinite number of distinct possible plans, each employing a different number of instances of $op_2$, and each potentially returning a different set of tuples. Note that there is no solution to this problem using the definition of [25]; if there were it would involve no more than two operators.

Although it is unfortunate that our definition of solution does not afford a bound on plan length, we believe that it is the best definition. For example, under Rajaraman *et al.*'s definition the *userid-room, finger* plan would not be a solution to *query-for-first-names* because the implication is not bidirectional. To see this not that the plan only finds the first names of people *who have email addresses*, not all people in the office. The bidirectional definition is an attempt to encode the fact that a plan should return *all* information, but we believe this to be fundamentally unacheivable because most information sources are incomplete. Hence bidirectional implication gains nothing.

# References


[1] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, August 1990.

[2] Yigal Arens, Chin Chee, Chun-Nan Hsu, Hoh In, and Craig Knoblock. Query processing in an information mediator. In *Proceedings of the*

*ARPA / Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, Tucson, AZ, 1994.

[3] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[4] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[5] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of IPSJ Conference*, 1994.

[6] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, June 1994.

[7] O. Etzioni, K. Golden, and D. Weld. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 178–189, San Francisco, CA, June 1994. Morgan Kaufmann.

[8] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, San Francisco, CA, October 1992. Morgan Kaufmann. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[9] O. Etzioni and D. Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, July 1994. See `http://www.cs.washington.edu/research/softbots`.

[10] Oren Etzioni and Daniel Weld. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert*, pages 44–49, August 1995.

[11] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

[12] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. on A.I.*, pages 1048–1054, Menlo Park, CA, July 1994. AAAI Press.

[13] A. Gupta, Garcia-Monlina H., J. Ullman, and Y. Papakonstantinou. A query translation scheme for rapid implementation of wrappers. In *Database and knowledge-base systems*, volume 1. Computer Science Press, 1989.

[14] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The information manifold. In *Working Notes of the AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pages 85–91, Stanford University, 1995. AAAI Press. To order a copy, contact sss@aaai.org.

[15] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. 15th Int. Joint Conf. on A.I.*, pages 1686–1693, 1995.

[16] Craig Knoblock and Alon Levy, editors. *Working Notes of the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, Stanford University, 1995. AAAI Press. To order a copy, contact sss@aaai.org.

[17] K. Krebsbach, D. Olawsky, and M. Gini. An empirical study of sensing and defaulting in planning. In *Proc. 1st Intl. Conf. on A.I. Planning Systems*, pages 136–144, June 1992.

[18] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieval*, 5 (2), September 1995.

[19] S. Minton, J. Bresina, and M. Drummond. Commitment strategies in planning: A comparative analysis. In *Proceedings of IJCAI-91*, pages 259–265, August 1991.

[20] S. Minton, M. Drummond, J. Bresina, and A. Phillips. Total order vs. partial order planning: Factors influencing performance. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992.

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

[22] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, pages 867–874, 1987.

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

[24] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. on A.I. Planning Systems*, pages 189–197, June 1992.

[25] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Principles of Database Systems*, 1995.

[26] G. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier, New York, 1975.

[27] J. Ullman. Database and knowledge-base systems. In *Database and knowledge-base systems*, volume 1. Computer Science Press, 1988.

[28] J. Ullman. Database and knowledge-base systems. In *Database and knowledge-base systems*, volume 2. Computer Science Press, 1989.

[29] D. Weld. The role of intelligent systems in the national information infrastructure. *AI Magazine*, 16(3):45–64, Fall 1995.