Estimated-Regression Planning for Interactions with Web Services [*]

Drew McDermott
Yale University Computer Science Department

Abstract

"Web services" are agents on the web that provide services to other agents. Interacting with a web service is essentially a planning problem, provided the service exposes an interface containing action definitions, which in fact is an elegant representation of how web services actually behave. The question is what sort of planner is best suited for solving the resulting problems, given that dealing with web services involves gathering information and then acting on it. *Estimated-regression* planners use a backward analysis of the difficulty of a goal to guide a forward search through situation space. They are well suited to the web-services domain because it is easy to relax the assumption of complete knowledge, and to formalize what it is they don't know and could find out by sending the relevant messages. Applying them to this domain requires extending classical notations (e.g., PDDL) in various ways. A preliminary implementation of these ideas has been constructed, and further tests are underway.

The Solution and the Problem

Estimated-regression planning is the name given to a family of planners including Unpop (McDermott 1996; 1999) and HSP (Bonet, Loerincs, & Geffner 1997; Bonet & Geffner 2001), in which situation-space search is guided by a heuristic estimator obtained by backward chaining in a "relaxed" problem space. Typically the relaxation neglects interactions, both constructive and destructive, between actions that achieve goals, and in particular neglects deletions completely. The resulting space is so much smaller than situation space that a planner can build complete representation of it, called a *regression graph*. The regression graph reveals, for each conjunct of a goal, the minimal sequence of actions that could achieve it.[1]

Estimated-regression planners have been applied to classical-planning domains, those in which perfect informa-

tion is assumed, but one of the reasons why these planners are interesting is that it seems they might be generalizable to problems that violate classical assumptions. An example is the domain of "web services," a phrase that denotes agents living on the World-Wide Web that provide services of various sorts in response to the appropriate messages. A typical web service might sell microprocessors in bulk to computer manufacturers. A *client* agent might send its order, plus some payment information, to the *service* agent, which sends back either a confirmation number or a failure message of some sort. More than one message exchange is typically necessary; for instance, the client might need to get a product-id number from the service in order to construct its order.

For this scenario to work out, the client must know what messages the service expects and what replies it produces. Because services are detected dynamically, each service must expose its interface information in a formal notation so that agents can know what they expect. There are differing models of what it means to be "formal" in this context, but one obvious idea is that what the service exposes is a set of primitive actions and canned interaction plans, which the client views as a planning domain. It constructs a plan for interacting with the service, executes it, and then takes stock. If the plan has failed, the client has at least gained some information, which it can use in the next cycle of planning and execution.

The planning problems that will naturally occur are not classical, for the obvious reason that classical-planning domains assume complete information, and many of the actions an agent performs on the Web are for the purpose of gaining information. However, in one respect the web-service problem fits the classical Strips-style paradigm very nicely: actions have preconditions and effects that are expressible as simple propositions. That means that it is easy to compute the contribution an action would make toward a goal, i.e., whether it would achieve or undo a piece of it, and what prior condition would be necessary in order for it to have that effect. This computation is what we mean by the term *regression*.

To apply an estimated-regression planner to this problem area, we have to deal with several issues: How can we model gaining information? What does it mean to solve a planning problem in circumstances where few plans are guaranteed to

[1]I intend the term "regression graph" to cover the data structures constructed by all the planners in this family; in my previous work I have used the term *regression-match graph,* to emphasize that my planner defers variable instantiation until the graph is constructed, and I use this term below when the distinction is important.

succeed? Can canned plans and loops be incorporated into a "quasi-classical" planner? In this paper, I address the first of two of these issues, although I believe that progress can be made on the third as well.

Related Work

One could write a large survey paper on all the work related to the research reported here. The industrial community is busily at work developing standards for describing web services. The UDDI ("Universal Description, Discovery, and Integration") Consortium is developing standards for registering web services. (See `http://www.uddi.org`.) This includes systems for describing interactions with web services such as WSDL (Christensen *et al.* 2001), and XLANG (Thatte 2001). WSDL mainly describes the contents of messages to and from agents at an abstract level, plus information on how the contents are mapped to more concrete protocols. XLANG is a language for writing scripts with which to deal with web services. My focus in this paper is in *creating* such scripts from more primitive action descriptions, but where appropriate canned scripts could be very helpful. Comparison of XLANG with the action language of PDDL (McDermott 1998) would be an interesting exercise.

There is a tremendous amount of research on agents for gathering information on the Web. Some focuses on extracting information from web pages of disparate types (Knoblock *et al.* 2001; Kushmerick 2000). Much of it focuses on efficient query planning (Ullman 1997; Ambite *et al.* 2001; Friedman & Weld 1997). This work is in a more specialized realm than I am considering here, but would play a role in dealing with web services if large-scale information gathering became an important part of the web-service problem.

The work on information gathering ties in with methods for relaxing the traditional closed-world assumptions of AI systems (Levy 1996; Etzioni, Golden, & Weld 1997), by stating "local closed-world assumptions," and using them to decide when to stop looking for information. I take the approach of explicitly reasoning about what an agent knows or does not know, which is probably equivalent to local closed-world assumptions in the end.

The work most closely related to mine is the work on *contingent planning*, in which planners try to take into account the need to insert sensing and branching steps into plans. The earliest work in this area is (Warren 1976). The area was revived by the work of (Peot & Smith 1992) on contingent partial-order planning. The work closest to mine in spirit is (Pryor & Collins 1996; Pryor 1995), in which observations are supported specifically to support decisions. The only work on contingent planning for estimated-regression planners is (Geffner 1998; Bonet & Geffner 2000). It takes a rather different approach in which the world is modeled as a Markov process, totally or partially observable. That model doesn't fit the web-service domain too well.

Domain Formalization

The central issue in formalizing a web service is how to represent the transmission of information. Other actions can be represented using standard techniques. For instance, if it is necessary to log in to a service before proceeding, we could express this in PDDL as

```
(:action login
    :parameters (a - Agent pw - String)
    :precondition (password pw a)
    :effect (logged-in a))
```

We should pause here to point out that on the Web this formalism must be encoded in XML, and that encoding it in DAML/RDF would be even better, since it would then become transparent to a wider variety of agents. For details on how to do that encoding, see (McDermott & Dou 2001). However, in this paper we will neglect the XML/DAML/RDF layer of abstraction almost completely.

Plan-Step Values

PDDL lacks the ability to specify that performing an action causes certain information to be created or learned. For example, one might want the action of sending a message to an agent A to generate a *message id* that A can use in a future reply to indicate the message that it is responding to. The existence of this id is not really an `:effect` in the PDDL sense, and even if it were it would be awkward to state exactly what kind of effect it was. Instead, it can be considered to be the *value* of the action, a value that can be passed on as an input to later steps of a plan.

The need to pass information from plan step to plan step has been known for a long time (McDermott 1978). Many researchers have used the Prolog model, in which goal sequences have variables, some of which acquire values as execution progresses. In the sequence (`look-for-people ?room ?people`), (`tell ?people (leave)`) we might suppose that `?room` has a value at the beginning, that `?people` gets set by the first step, and then serves as an input to the second. Sometimes the variables are flagged to indicate when they get set (Etzioni *et al.* 1992; Levesque *et al.* 1997). For information-gathering agents the flags serve to indicate which variables must be set in order for an information server to be able to process a query.

The Prolog model makes sense for information-gathering agents, but seems like a poor fit to regular planners. Prolog can be given a clean semantics in which all variables are interpreted as universally quantified. During execution of a Prolog program variables become steadily more constrained by unification, but the concept of a variable being "set" is meaningless (because the point where a variable's value becomes variable-free has no particular significance). When a failure occurs, the program backtracks, undoing some of the constraints on variable values. *None* of these features apply to planning, where plan steps have actual effects on the world, yield well-defined nuggets of information, and cannot be undone just by backtracking.

Instead we opt for a model in which steps have an optional `:value` field that describes the value it returns. In the simplest case this description is a type. If the field is absent, the type defaults to "no values." So our message-id example might look like this:

```
(:action send
    :parameters
        (?agt - Agent ?sent - Message)
    :value Message-id
    :precondition (web-agent ?agt)
    :effect (rep
        ?agt (step-value this-step)
        ?sent))
```

We introduce two new symbols. `(step-value s)` is the value returned by a step in a plan. For example, an agent might send a message, then transmit the corresponding "reply-to" id to another agent that will actually process the reply:

```
(series (tag step-1
             (send merrill-lynch
                   (buy ibm 100) logger))
        (send logger (step-value step-1)))
```

where `logger` is the agent we want merrill-lynch to talk to about this transaction from now on.

The symbol `this-step` in action definitions is a placeholder for the actual step id that will be inserted when the action is instantiated.

Step values can have fairly complex types, such as "a tuple consisting of a string and a list of street addresses." For that and other reasons, we must extend PDDL with a polymorphic type system (Cardelli & Wegner 1985). That would allow us to declare that the `:value` of a step was of type `(Tup String (Lst Street-address))`. Then we could refer to the second element of the value of `step-3` by writing `(!_<2> (step-value step-3))`. Because this is fairly opaque, we supply some syntatic sugar. In the definition of this hypothetical action we could write

```
(:action fetch-data
    ...
    :value
        (Tup label - String
             addrs
                  - (Lst Street-address))
    ...)
```

and if `step-3` is an instance of `fetch-data`, we can refer to its second component by writing `(!_addrs (step-value step-3))`. Furthermore, we can say `addrs` instead of `(!_addrs (step-value this-step))` in the `:effect` field of `fetch-data`. With these conventions, the `send` action can be defined as

```
(:action send
    :parameters
        (?agt - Agent ?sent - Message)
    :value (id - Message-id)
    :precondition (web-agent agt)
    :effect (reply-pending agt id sent))
```

In the enhanced language, which is called Opt, question marks on bound variables are optional, and they have been dropped in this example. It seems stylistically clearer to reserve question marks for free variables. Here

the `:parameters` and `:value` fields serve as variable binders.

A key point about the notation is that the `:value` field of an action definition just specifies the type of the value the action generates. The *meaning* of the value is given by the `:effect` field. Further examples will appear below.

Incomplete Knowledge

Most planners make a "closed-world assumption" at some point, assuming that if an atomic formula is not explicitly noted as true (or straightforwardly deducible) in a situation, it may be assumed to be false. In the context of web-service planning, the planner typically knows what it doesn't know; if it is planning to buy something from a service, it doesn't know what the price is until it asks. We formalize this idea with a predicate `(know-val X)`. Some care is required in saying what X is exactly. Consider the statement `(know-val (price bogotron))`. One might suppose that `price` was a function from products to monetary values, in which case from the knowledge that `(price bogotron)` was an integer between $95 and $105, the planner could infer

```
(or (know-val $95)
    (know-val $96)
    ...
(know-val $105))
```

which makes no sense. It's logically more accurate to consider X to be a *learnable term,* an object of type `(Learnable Money)` in this case. We also have a function `(val X)` which denotes the value of such a term. The abbreviation `(know-val-is X v)` is synonymous with `(and (know-val X) (= (val X) v))`, but is easier for the planning algorithm to cope with.

There is an action `(compute term)` which fails if *term* contains learnable subexpressions whose values are not known. Otherwise, it succeeds, and its value is the value of *term*, with all learnable subexpressions replaced by their values. In practice, the value of a learnable expression is a purely computational entity such as an integer, a string, or a tuple of purely computational entities. (But we don't need to take a position on what constitutes a purely computational entity.)

Note that once we have the `know-val` predicate, we can reinstate the closed-world assumption. It may be that the planner doesn't know the value of `(< (share-price ibm) (dollar 100))`, but then it knows that it doesn't know, because `(know-val (< (share-price ibm) (dollar 100)))` is not deducible.[2]

Finally, we have an action `(test X P_T P_F)`. It computes X, a Boolean expression, in the sense described above. If the result is `true`, action P_T is executed, else P_F.

---

[2] Technically, we should replace `<` with a variant that operates on learnable numbers instead of numbers. However, it would merely clutter this paper to do that. So I will pretend that Opt allows overloading of functions, which it does not.

Planning Algorithm

I have extended the Unpop planner (McDermott 1996; 1999) to handle simple web-service problems. The resulting planner is called Optop ("Opt-based total-order planner"). Here is a brief review of how Unpop works: Given a goal and an initial situation, it does a situation-space search for a sequence of steps that achieve the goal. A search state is a series of feasible steps starting in the initial situation and culminating in a situation that is hopefully closer to satisfying the goal. We call this the *current situation* of the state. The search is guided by the following heuristic: For each search state, build a *regression-match graph* from the goal back to the current situation of that search state. The graph is essentially an AND/OR tree in which AND nodes correspond to the preconditions of an action (all of which must be achieved), and an OR node corresponds to a set of actions (one of which is sufficient to achieve a condition). The "tree" is actually a graph because there is just one occurrence of any given atomic formula, so that two AND nodes may share an element, and there may be a cycle from an AND node N to an atomic formula A in N, to an OR node that achieves A, and eventually to N again. The "leaves" of the resulting deformed tree are conditions that are either true in the current situation or are not in the add list of any action and hence cannot be further reduced (and are not achievable under any circumstances). An action is *recommended* if all of its preconditions are true in the initial situation. The planner considers taking only recommended actions next, and judges their cost by the total effort of the subtrees they occur in. The *(estimated) effort* of a subtree to achieve goal literal G is defined as 0 if G is true in the current situation, else the minimum over all actions A that achieve G of the cost of A + the sum of the efforts of the subgoals in the AND node built from the preconditions of A. (I apologize for the succinctness of this description; the gory details may be found in (McDermott 1999).)

Variables

In constructing the regression-match graph, a key question is how the preconditions of an action are turned into an AND node. Even if G does not contain variables, the preconditions of an action A that achieves G will in general contain some. If the planner leaves them in, then the accuracy of the graph is reduced, in two ways. First, when the preconditions are regressed (reduced through another AND node), more variables will appear, making the resulting literals less and less distinctive. The estimate of the difficulty of a literal such as (at toyota-1 pittsburgh-station) means more than the estimate of the difficulty of (at ?x ?y). Second, if two preconditions share a variable, summing their efforts loses accuracy, because what we really care about is the effort of achieving instances of both preconditions in which shared variables have the same value.

This problem is finessed in (Bonet, Loerincs, & Geffner 1997) by finding all possible ground instances of every action definition at the beginning of search for a plan. The Unpop planner solve the problem by matching the preconditions against the current situation, finding variable bindings

that make as many of the preconditions true in that situation as possible, leaving behind a residue of *differences* (Ernst & Newell 1969). If a difference contains variables, then at that point it is replaced by all instances obtained by substituting constants of the appropriate types for those variables. This process is called *maximal matching*. (For details, see (McDermott 1999).)

Unfortunately, neither of these strategies will work in the domain of web services, where many data types are infinite. For instance, there is no way to substitute all possible objects of type Message for a variable.

For this reason, we have to remove the assumption that atomic formulas in the regression-match graph are variable-free. The maximal matcher in Optop tries hard to eliminate preconditions and variables by matching and substitution, but sometimes it is left with differences containing unremovable variables.

It also simplifies the statement of complex domains if we allow the planner to backchain through axioms. That is, if there is a goal G, and an axiom (<- G' A), where G and G' unify with substitution θ, the maximal matcher will consider replacing G with $\theta(A)$. This step may introduce new variables, but they are treated the same as those obtained by goal regression.

The main complication that variables introduce is a consequence of the problem described above. To combine estimates of the costs of achieving the preconditions of an action into an estimate of the cost of achieving a conjunction of them, the planner must find consistent combinations of preconditions, that is, combinations that assign the same values to shared variables.

Ignorance and Branching

Recall that Unpop and Optop both work by running an inner loop of regression-graph construction inside an outer loop of search through situations (or, more precisely, step sequences leading to situations). The outer loop accepts the actions recommended by the graph constructor, and tries projecting each one to produce a new situation. Two things can happen:

- The new situation is identical to a previously encountered situation. In this case, the new one is discarded.

- The new situation has not been seen before, and so becomes the current situation of a new step sequence, which goes on the search queue for eventual further investigation.

In the process of creating the new situation, the projector must handle effects that mention (step-value this-step) in a special way. The projector first creates a new step with a new step id, whose external form is (tag *id action*). Occurrences of this-step in *action* are replaced by *id*. The projector must then decide what to do about the step-value. In some cases there may be a "normal" value. For instance, if an agent submits a credit-card number for authorization, the normal response it expects back is "ok." In this case, it can replace (step-value *step-id*) with the normal value, but add a new step

```
(verify (= (step-value step-id)
           normal-value))
```

All `(verify P)` does is compute P and see if it comes out `true`. If so, plan execution continues; if not, the plan fails. (See below.)

If a step's value is just passed to another step, as in our example of the logger agent, there is no need for the planner to know the value at planning time. The planner does need to know the value during the regression-graph construction phase when a precondition occurs that contains a subterm of the form `(val X)`, where X is a learnable term. A precondition is a proposition, such as

```
(< (val (price widgex bogotron))
   (dollar 20))
```

Such a precondition cannot be satisfied without knowing the price Widgex charges for bogotrons.[3] So the graph builder inserts an AND node with action

```
(verify (< (val (price widgex
                      bogotron))
        (dollar 20)))
```

in the graph, with a single precondition `(know-val (price widgex bogotron))`. If this part of the graph yields a promising direction for the planner to pursue, then eventually the `verify` action will get added to the plan.

Verification actions can of course fail, and, except in the case of relying on "normal step values," often do. But Optop stays the course, and assumes that any `verify` step will succeed. This means that the outer loop of situation-space search is essentially the same as in Unpop. The planner lengthens plans until it finds one that achieves the goal it was given, or until it exceeds some resource bound. In the former case, if the plan contains no `verify` steps, then the planner has found a viable plan. In the latter case, if the planner never examined a plan containing a `verify` step, then the planner has failed to find a plan (which may be the correct outcome, if no plan exists).

The interesting cases are where a successful plan is found, but it contains `verify` steps; and where no plan is found, but at least one partial plan was investigated that had a `verify` step. The second case sounds like it would be very difficult to deal with. The planner may have looked at hundreds of partial plans, and if a `verify` was inserted early, many of them will contain `verify` steps. However, I believe the planner can just give up at this point. As we will discuss shortly, what we want to know about a `verify` step is, What would have happened if it had turned out otherwise? Presumably the answer we hope for is, The planner would have found another way to achieve its goal. For instance, in an example discussed by (Pryor & Collins 1996), if there are two roads to a destination, and one is blocked,

_____

[3]I'm assuming that there is no way for the planning agent to change the price Widgex charges. If it does, we would have to make the price be a "fluent" as discussed in (McDermott 1998; 1999), and other options would open up. I'm also neglecting other possibilities, such as finding out that the price is less than $20 by finding out that it is less than $10.

the agent should try taking the other. But in an estimated-regression planner, this second alternative is already being entertained from the very beginning. It's in the regression-match graph, but just doesn't look as attractive as taking the first route. If the planner fails to find a solution assuming the first route is open, then it will eventually consider taking the second route, not because the first alternative is unavailable, but because the first alternative didn't lead to a competitive plan. That is, it considers taking the second route without even looking to see if the first is blocked. But we started with the assumption that after all this activity the planner never could solve the problem. In that case, it seems very unlikely that it could solve it with the added assumption that the first route is blocked.

Hence the only case we really need to consider is where the planner has found a successful plan, but the plan relies on `verify`ing things that aren't necessarily true. Note that the argument of the previous paragraph implies that this plan is in a sense "optimistic." The propositions the planner chose to verify are the ones that would lead to a short action sequence in the opinion of the regression-match graph (an opinion that isn't always reliable, of course). Now the planner has to go back and consider what happens if optimism doesn't apply. It finds the first `verify` in the "successful" action sequence, and considers what it would mean if that step failed. What it would mean is that the proposition being verified is false. The `verify` step is reprojected, yielding a new situation. Starting from this situation, Optop calls itself recursively with the original goal. The result is an alternative plan (we hope). The original plan, of the form `(series $S_1$ (verify $P$) $S_2$)`, and the new one, `(series $S_3$)`, are combined into

```
(series S1
        (test P
              S2
              S3))
```

If the recursive call fails to find a plan, then S_3 will just be `(fail)`. This is not the show-stopper it would be in the classical world. As previous researchers have pointed out (Peot & Smith 1992; Pryor 1995), in the presence of contingencies it's hard to guarantee success, and not really necessary.

So far the planner has dealt with just the first contingency in the sequence. It could continue to find further `verify` steps downstream and convert them as it did the first. However, beyond a certain point it is counterproductive to explore every contingency. Many `verify`'s should just be left in, deferring planning until they actually fail.

Implementation

A preliminary implementation of Optop has been written. Most of the work has gone into allowing variables into the regression-match graph, allowing goal reduction through axioms, and getting right the detailed implications of having `step-value` and `this-step` in the language.

Figure 1 shows a domain `www-agents` in which Optop can find simple plans. There are just two actions, `send` and

```
(define (domain www-agents)
    (:extends knowing regression-planning commerce)
    (:requirements :existential-preconditions :conditional-effects)
    (:types Message - Obj Message-id - String)

    (:functions (price-quote ?m - Money)
                (query-in-stock ?pid - Product-id)
                (reply-in-stock ?b - Boolean)
                - Message)

    (:predicates (web-agent ?x - Agent)
                 (reply-pending a - Agent  id - Message-id msg - Message)
                 (message-exchange ?interlocutor - Agent
                                   ?sent ?received - Message
                                   ?eff - Prop)
                 (expected-reply a - Agent sent expect-back - Message))

    (:axiom
        :vars (?agt - Agent ?msg-id - Message-id
               ?sent ?reply - Message)
        :implies (normal-step-value (receive ?agt ?msg-id)
                                    ?reply)
        :context (and (web-agent ?agt)
                      (reply-pending ?agt ?msg-id ?sent)
                      (expected-reply ?agt ?sent ?reply)))

    (:action send
        :parameters (?agt - Agent ?sent - Message)
        :value (?sid - Message-id)
        :precondition (web-agent ?agt)
        :effect (reply-pending ?agt ?sid ?sent))

    (:action receive
        :parameters (?agt - Agent ?sid - Message-id)
        :vars (?sent - Message ?eff - Prop)
        :precondition (and (web-agent ?agt)
                           (reply-pending ?agt ?sid ?sent))
        :value (?received - Message)
        :effect (when (message-exchange ?agt ?sent ?received ?eff)
                      ?eff)))
```

Figure 1: The www-agents domain

`receive`, and the effect of `receive` depends entirely on an assertion of the form

```
(message-exchange sendee
                  message-sent
                  reply-received
                  effect)
```

For example:

```
(freevars (s - Merchant
           pid - Product-id
           b - Boolean)
   (message-exchange
       ?s
       (query-in-stock ?pid)
       (reply-in-stock ?b)
       (know-val-is (in-stock ?s ?pid)
                    ?b)))
```

Given the problem shown in figure 2, Optop arrives at the plan

```
(series
    (tag step-13
        (send amazin.com
              (query-in-stock
                  (prodid "p001"))))
    (tag step-14
        (receive amazin.com
                 (step-value step-13)))
    (verify (= (step-value step-14)
               (reply-in-stock true))))
```

That is, to know that Amazin.com has a certain book in stock, it suffices to ask them; they probably do.

This is not an earth-shaking result, nor is the fact that Optop had to do no search at all to arrive at it. However, the ability to find long plans in tricky domains is not the main focus of this paper. My purpose is to show that estimated-regression planners are well suited to planning to deal with web services.

Conclusions and Future Work

The conclusions I would like to draw from this work are these:

- Web services are an interesting domain for planning research, since they fit the classical model of discrete actions, but require eliminating closed-world assumptions and allowing for branching plans.

- Representations such as PDDL must be extended somewhat to capture the information required for formalizing web services. Because complex and diverse data structures must be sent across the internet, we need a more robust type notation to express the content of messages. Actions need a `:value` field to express the types of the data they return. These notations can all be made to mesh nicely.

- Estimated-regression planners seem well suited to working on the resulting class of problems. They main change

is to integrate the formalization of the `know-val` predicate into the regression-graph-building and projection machinery. The key idea is that for a computational agent to know the value of something is for it to have a purely computational term that denotes that value.

- A natural way to generate branching plans is to generate linear plans and then add branches where assumptions might be false.

There is an enormous amount of work left to do, besides bringing the implementation up to the point where it matches the description above.

Dealing with web services inevitably requires loops. For instance, to find the lowest price for a product requires sending messages to all the known providers of that product. Loops are radically unclassical. I believe the right approach to this problem is to think about how to integrate the use of canned plans into a classical planner. One way is allow the planner to add a complex action to the end of an evolving plan, not just a primitive action. The complex action then becomes a resource for further plan growth, because it suggests the next thing to do without any search at all. These suggestions must, however, compete with actions suggested by the regular regression-estimation process. Of course, this idea will in the end produce only "unrolled" loops, unless it is combined with a mechanism for deferring some planning until after execution has revealed more information.

One problem that is not a problem at all in this framework is the problem of *multiagent composition*, which arises because scripting approaches such as XLANG (Thatte 2001) and DAML-S (Ankolenkar *et al.* 2001) focus entirely on an interaction between two agents. From the point of view espoused here, all that is required is to merge the descriptions of the agents your agent is considering talking to, then have your planner solve the problem in the enlarged world. Talking to two or more agents has exactly the same logical status as talking to one.[4]

References

Ambite, J. L.; Knoblock, C. A.; Muslea, I.; and Philpot, A. 2001. Compiling source descriptions for efficient and flexible information integration. *J. Intelligent Information Systems* 16(2):149–187.

Ankolenkar, A.; Burstein, M.; Hobbs, J.; Lassila, O.; Martin, D.; McIlraith, S.; Narayanan, S.; Paolucci, M.; Payne, T.; Sycara, K.; and Zeng, H. 2001. Daml-s: A semantic markup language for web services. In *Proc. Semantic Web Working Symposium*, 411–430.

Bonet, B., and Geffner, H. 2000. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. 5th Int. Conf. on AI Planning and Scheduling (AIPS-00)*. AAAI Press.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2).

---

[4]Of course there are issues that arise in reconciling ontological differences between the agents. See (McDermott, Burstein, & Smith 2001).

```
(define (problem bb-prob1)
   (:domain www-agents)
   (:objects amazin.com wabash.com - Merchant)
   (:facts (web-agent amazin.com)
           (web-agent wabash.com)
           (freevars (pid - Product-id)
              (expected-reply amazin.com
                             (query-in-stock ?pid)
                             (reply-in-stock true)))
           (freevars (sid - Message-id pid - Product-id)
              (expected-reply wabash.com
                             (query-in-stock ?pid)
                             (reply-in-stock false))))
   (:goal (know-val-is (in-stock amazin.com (prodid "P001"))
                      true)))
```

Figure 2: Problem in the `www-agents` domain

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A fast and robust action selection mechanism for planning. In *Proc. AAAI-97*.

Cardelli, L., and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17(4):471–522.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2001. Web Services Definition Language (Wsdl) 1.1. Technical report, W3C. , available at `http://www.w3c.org/TR/wsdl`.

Ernst, G. W., and Newell, A. 1969. *GPS: A Case Study in Generality and Problem Solving*. Academic Press.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. Third International Conf. on Knowledge Representation and Reasoning*, 115–125. Morgan Kaufmann.

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

Friedman, M., and Weld, D. S. 1997. Efficiently executing information-gathering plans. In *Proc. Ijcai-97*.

Geffner, H. 1998. Classical, probabilistic, and contingent planning: three models, one algorithm. Workshop on Planning as Combinatorial Search: Propositional, Graph-Based, and Disjunctive Planning Methods.

Knoblock, C. A.; Minton, S.; Ambite, J. L.; Ashish, N.; Muslea, I.; Philpot, A. G.; and Tejada, S. 2001. The Ariadne approach to web-based information integration. *Int. J. Cooperative Information Systems (IJCIS)* 10(1-2):145–169. Special Issue on Intelligent Information Agents: Theory and Applications.

Kushmerick, N. 2000. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence* 118(12):15–68.

Levesque, H.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *J. Logic Programming* 31:59–84.

Levy, A. Y. 1996. Obtaining complete answers from incomplete databases. In *Proc. 22nd Conference on Very Large Databases*.

McDermott, D., and Dou, D. 2001. Embedding logic in daml/rdf. . Available at `http://www.cs.yale.edu/d̃vm`.

McDermott, D.; Burstein, M.; and Smith, D. 2001. Overcoming ontology mismatches in transactions with self-describing agents. In *Proc. Semantic Web Working Symposium*, 285–302.

McDermott, D. 1978. Planning and acting. *Cognitive Science* 2(2):71–109.

McDermott, D. 1996. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems*, 142–149.

McDermott, D. 1998. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science. (CVC Report 98-003).

McDermott, D. 1999. Using Regression-match Graphs to Control Search in Planning. *Artificial Intelligence* 109(1–2):111–159.

Peot, M., and Smith, D. 1992. Conditional nonlinear planning. In Hendler, J., ed., *Proceedings of the First International Conf. on AI Planning Systems*. 189–197.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *J. of Artificial Intelligence Research* 4:287–339.

Pryor, L. 1995. Decisions, decisions: knowledge goals in planning. In *Hybrid problems, hybrid solutions (Proc. AISB-95)*, 181–192. J. Hallam (ed) IOS Press.

Thatte, S. 2001. Xlang: Web services for business process design. At http://www.gotdotnet. com/team/xmlwsspecs/xlang-c/default.htm

Ullman, J. D. 1997. Information integration using logical views. In *Proc. 6th Intl. Conf. on Database Theory (ICDT-97), Lecture Notes in Computer Science*, 19–40. Springer-Verlag.

Warren, D. H. 1976. Generating conditional plans and programs. In *Proc. AISB Summer Conf*, 344–354.