Proceedings of the 2007 IEEE Symposium on
Computational Intelligence and Data Mining (CIDM 2007)

Reactive Query Policies: A Formalism for Planning with Volatile External Information

Tsz-Chiu Au
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: chiu@cs.umd.edu

Dana Nau
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: nau@cs.umd.edu

***Abstract*— To generate plans for collecting data for data mining, an important problem is *information volatility during planning*: the information needed by the planning system may change or expire during the planning process, as changes occur in the data being collected. In such situations, the planning system faces two challenges: how to generate plans despite these changes, and how to guarantee that a plan returned by the planner will remain valid for some period of time after the planning ends.**

The focus of our work is to address both of the above challenges. In particular, we provide:

1) **A formalism for *reactive query policies*, a class of strategies for deciding when to reissue queries for information that has changed during the planning process. This class includes all query management strategies that have yet been developed.**
2) **A new reactive query policy called the *presumptive* strategy. In our experiments, the presumptive strategy ran exponentially faster than the lazy strategy, the best previously known query management strategy. In the hardest set of problems we tested, the presumptive strategy took 4.7% as much time and generated 6.9% as many queries as the lazy strategy.**

I. Introduction

One difficulty with integrating planning and data mining is that the results of data collection and analysis may influence the planning process itself. To decide how best to plan for subsequent data-collection and analysis efforts, the planner may need to request information from external information sources such as sensors, databases, data analysis programs, web services, and the like, incurring a lag time for receiving the answers. Furthermore, the planning activity may occur over a period of several hours or even several weeks [1]–[4]. This forces the planner to deal with *information volatility during planning*: as changes occur in the external world, the information needed by the planning system may change or expire before the planning process completes.

AI-planning researchers have not paid much attention to information volatility during planning. Instead, their research has concentrated on static environments in which no changes occur in the world other than the ones caused by executing the planner's plans. But it is easy to find many practical situations in which information volatility occurs. For instance:

- For analyzing large amounts of data in data mining and other applications, grid computing is an increasingly important technique [5]. In grid- and utility-based computing applications [6], one might want to reserve computing resources owned by several different companies though some grid services on the Web to accomplish a computational task, but the availability and the amount of the resources will keep changing.
- When collecting and analyzing data from web services, one problem is that the information can change very frequently. In [3] and [4], Kuter et al. describe a domain-specific system that uses AI planning to do web service composition; this system was explicitly designed to change its plans in response to information volatility.
- Readers who have tried to make travel plans will probably recognize the kind of screenshot in Figure 1. Here, the information about an airline flight expired while one of us was trying to plan some other details of a trip.

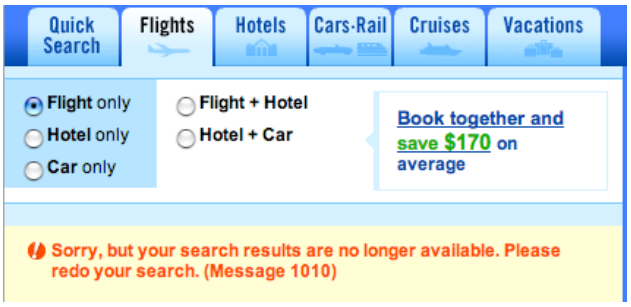

Fig. 1. A screenshot of an online airline-ticket reservation system. The prices expired while the user was trying to plan other details of the trip.

In such environments, how to cope with changes of external information and at the same time generate a plan that can be executed correctly is a big challenge. Most existing planners will not do this correctly unless they are modified.

In some cases, the interleaving of planning and execution [7] is a good strategy to deal with volatile information. But if the wrong choice of action can cause a failure that is irrecoverable (or recoverable only at a large cost), then it is necessary for the planning system to reason, while the plan is being generated, about whether the assumptions that are being used to choose an action will still be true when the action is executed.

To the best of our knowledge, our previous work [8] was the first to provide a way for planning systems to do such reasoning. A primary limitation of that work was that although

it provided plans that were guaranteed to be correct at the time they were returned by the planner, there was no guarantee that such plans would be correct even a millisecond later. In this paper, we examine how to produce plans that are guaranteed to remain valid for some period of time T after they are returned.

In another previous paper [9], we did a theoretical study of whether a planner in a volatile external information environment can ever be complete. More precisely, we provided an incompleteness theorem showing that it is, in principle, impossible to have a planner that can successfully find a valid solution in every solvable environment.[1]

There is no way that can get around with the incompleteness without modifying the underlying assumptions of the problem. However, if we are willing to give up the requirement that a planner be complete, then it is possible to design planners that have a high probability of returning a solution in practical planning problems. This is the direction that this paper pursues. In particular, our contributions are:

1) We provide a formalism for a large class of query management strategies (i.e., strategies for when to reissue queries for information that changes during the planning process) that we call *reactive query policies*, by representing these strategies as collections of event-driven rules. We state the condition under which a reactive query policy can return plan would remain valid for a given period of time. This class includes all query management strategies that have appeared in the published literature.
2) We provide new reactive query policy called the *presumptive* strategy. In our experiments, the presumptive strategy ran exponentially faster than the lazy strategy, the best previously known query management strategy [8]. In the hardest set of problems we tested, the presumptive strategy took only 4.7% as much time and generated only 6.9% as many queries as the lazy strategy.

II. Problem Definition

In this section, we define planning problems that require planners to gather information from external information sources in order to accomplish the planning task. Then we describe a way to turn any planner into one that can handle external information. Finally, we define a model of volatile external information, and the information-gathering planning problems whose external information are volatile.

A. Planning with External Information

In the AI planning literature, it is conventional to assume that a planner begins with a complete description of the world, and that the world remains static while the planner is running. We will use the word *conventional* to refer to these planning problems, the languages in which they are written, and the planners that solve them. Depending on the type of problem in question, a *solution* to a conventional planning problem can be either a plan or a policy. A conventional planning problem P can have more than one solution; let $solution(P)$ be the set of all solutions of P. A conventional planner $\mathcal{A}$ takes P as an input and generates a solution or a symbol Failure. We denote the output by $\mathcal{A}(P)$. If $\mathcal{A}$ is *complete*, then either (1) $\mathcal{A}(P) \in solution(P)$ if $solution(P)$ is not an empty set, or (2) $\mathcal{A}(P) = \mathsf{Failure}$ if $solution(P)$ is an empty set.

Let L be a conventional planning language such as PDDL [11]. We'll construct a new language $\hat{L}$ that contains all the symbols of L plus a finite set $\hat{U}$ of additional symbols called *unknowns*. For each unknown $u \in \hat{U}$, there is a set $dom(u)$ of terms in L. We call $dom(u)$ the *domain* of u, and the terms in $dom(u)$ the *values* of u. For any expression e in L, let $\mathsf{terms}(e)$ be the set of all terms in e. Then we construct a set $\hat{L}$ of expressions by taking any expression e in L and replacing zero or more terms in $\mathsf{terms}(e)$ with unknowns. More precisely, let $e[t_1/t'_1, t_2/t'_2, \ldots, t_k/t'_k]$ be the expression after replacing the term t_i in e with another term t'_i, for $1 \leq i \leq k$. Then the new language is $\hat{L} = \{e[(t_i/u_i)_{i..k}] : e \in L, \{t_1, t_2, \ldots, t_k\} \subseteq \mathsf{terms}(e), t_i \in dom(u_i)\}$.

For any expression $\hat{e} \in \hat{L}$, let $unknowns(\hat{e})$ be the set of all unknowns in $\hat{e}$. Then we say $\hat{e}$ is *EI-ground* ("EI" stands for "external information") if $unknowns(\hat{e}) = \emptyset$; otherwise it is *EI-unground*. Clearly, an EI-ground expression is in L. A EI-ground expression e is a *EI-instance* of $\hat{e}$ if e can be obtained from $\hat{e}$ by substituting values for the unknowns in $\hat{e}$. Thus, the set of all EI-instances of $\hat{e}$, denoted by $instances(\hat{e})$, is $\{\hat{e}[(u_i/v_i)_{i=1..k}] : u_i \in unknowns(\hat{e}), v_i \in dom(u_i)\}$.

Suppose $\mathcal{A}$ is a conventional planner whose problems are written in L. Let $\mathcal{P} \subseteq L$ be the set of all planning problems for $\mathcal{A}$. We extend $\mathcal{P}$ to $\hat{\mathcal{P}} = \{\hat{e} : \hat{e} \in \hat{L}, instances(\hat{e}) \subseteq \mathcal{P}\}$. The planning problems in $\hat{\mathcal{P}}$ are called *EI-problems*. For simplicity, we require that for any $\hat{P} \in \hat{\mathcal{P}}$, $instances(\hat{P}) \subseteq \mathcal{P}$.

A *possible solution* for a EI-problem $\hat{P} \in \hat{\mathcal{P}}$ is any solution for any EI-instance of $\hat{P}$. A *EI-planner* is a planner whose input is a EI-problem $\hat{P}$ and whose output is a possible solution for $\hat{P}$. A EI-planner can be constructed from a conventional planner $\mathcal{A}$ by a *EI-conversion* as follows: $\hat{\mathcal{A}}$ is exactly the same as $\mathcal{A}$ except it maintains every unknown symbolically in its *execution state*, an aggregation of all data in the memory, register, invocation stack, program counter, etc., at a particular moment of time of the execution of $\hat{\mathcal{A}}$. When $\hat{\mathcal{A}}$ tries to read the value of an unknown u for the first time during its execution, $\hat{\mathcal{A}}$ suspends itself, requests a value for u, waits until a value for u comes back, and then resume its execution. There are other type of planner-specific EI-conversions (e.g. ENQUIRER [3]) that might not require planners to wait for an answer after a query is made. However, these EI-conversions rely on the internal structure of the conventional planners, hence are not applicable to every conventional planner.

B. Planning with Volatile External Information

If external information is *static* (i.e., the values for the unknowns never change), all we need to solve a EI-problem is

[1]This theorem resembles the famous impossibility theorems in distributed systems (http://www.podc.org/influential/2001.html), which say that some problems in distributed systems are fundamentally unsolvable [10]. Remedying this problem is far from trivial, because it requires modifications to the fundamental assumptions of the distributed system model people have been used. Many of those solutions are not ideal, but they are needed in order to able to solve certain problems.

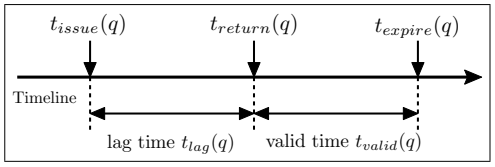

Fig. 2. The lag time and the valid time of a query q. $v(q)$ is guaranteed valid only between $t_{return}(q)$ and $t_{expire}(q)$. After $t_{expire}(q)$, the value may possibly change.

a EI-planner derived from a sound and complete conventional planner (by EI-conversion); otherwise, $\hat{\mathcal{A}}$ will not be sufficient by itself, because the substituted value in $\hat{\mathcal{A}}$'s execution state may change before the end of the planning process. Therefore, we need to develop a new kind of planners, which we'll call a *VEI-planner* (volatile external information planner), that can manage the planning process in reaction to changes in values. Like a EI-planner, a VEI-planner takes a EI-problem and returns a possible solution; but the VEI-planner's solution must satisfy certain volatile-information requirements as described below.

In our model of volatile external information, volatile information are time-dependent. For each unknown u, we assume there is a piecewise-constant function $\mathbb{V}_u : \mathbb{R}_{\geq 0} \rightarrow dom(u)$, which specifies the value of u at time t, and $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers. At any time t, we say $\mathbb{V}_u(t)$ is the *valid* value of u. At the beginning, a VEI-planner $\tilde{\mathcal{A}}$ knows nothing about $\mathbb{V}_u$; the only way for $\tilde{\mathcal{A}}$ to get to know the valid values of an unknown is to issue *queries* to an *information source* I_u, which is the sole authority to tell $\tilde{\mathcal{A}}$ about $\mathbb{V}_u$. There may be several information sources, and we assume $\tilde{\mathcal{A}}$ knows which information source it should send a query to.

Suppose $\tilde{\mathcal{A}}$ sends a query q for an unknown u to I_u at time $t_{issue}(q)$, and then I_u returns a value $v(q)$ for q at time $t_{return}(q) > t_{issue}(q)$. If $\tilde{\mathcal{A}}$ knows nothing about how long $v(q)$ will remain true, it cannot provide any sort of guarantee about the correctness of the plan it generates: the information may change during plan execution, invalidating the plan. Thus, $\tilde{\mathcal{A}}$ needs to have a lower bound on how long the information will remain true. Therefore, we assume that I_u's answer $ans(q)$ will include both $v(q)$ and an *expiration time* $t_{expire}(q) > t_{return}(q)$; that is, $ans(q) = (u, v(q), t_{expire}(q))$. As shown in Figure 2, $v(q)$ is guaranteed to be valid until $t_{expire}(q)$, after which time it may change.

$v(q)$ is said to *remain valid* until $t_{expire}(q)$, i.e., for any time t on or after $t_{return}(q)$ and before $t_{expire}(q)$, $v(q) = \mathbb{V}_u(t)$; for any $t' \geq t_{expire}(q)$, $\mathbb{V}_u(t')$ may be $v(q)$ or may be something different. The *lag time* of q is $t_{lag}(q) = t_{return}(q) - t_{issue}(q)$, which includes both the time I_u takes to process the query and the time that messages travel back and forth. The *valid time* of q is $t_{valid}(q) = t_{valid}(q) - t_{expire}(q)$.

A *VEI-problem* $\tilde{P}$ is a pair $\langle \hat{P}, T \rangle$, where $\hat{P}$ is an EI-problem and $T \in \mathbb{R}_{\geq 0}$ is a *validity guarantee*. Informally, a solution for $\tilde{P}$ is any solution for $\hat{P}$ that remains valid for a time $\geq T$. More formally, we define the *current* EI-instance of $\hat{P}$ at time t to be $ground_t(\hat{P}) = \hat{P}[(u_i/\mathbb{V}_{u_i}(t))_{i=1..m}]$, where $\{u_1, u_2, \ldots, u_m\} = unknowns(\hat{P})$. A possible solution π of $\hat{P}$ is *valid* at time t if $\pi \in solution(ground_t(\hat{P}))$. Also, π is *T-valid* at time t if π remains valid during the time interval $[t, t+T)$ (i.e., $\pi \in solution(ground_{t'}(\hat{P}))$ for $t \leq t' < t+T$). π is a *T-valid solution* for $\tilde{P}$ if and only if π is T-valid at the time the VEI-planner returns π.

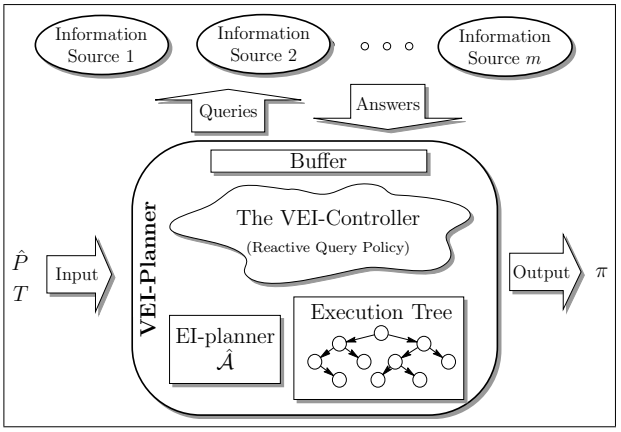

Fig. 3. Architecture for separate planning and query management. The VEI-planner takes a EI-problem $\hat{P}$ and a validity guarantee T, and generates a T-valid solution π. The planning task is conducted by a EI-planner $\hat{A}$, which can be obtained from a conventional planner by a EI-conversion.

A VEI-planner is a planner that takes a VEI-problem $\langle \hat{P}, T \rangle$ as an input and generates a possible solution π for $\hat{P}$. A VEI-planner is *successful* for $\langle \hat{P}, T \rangle$ if π is a T-valid solution.

III. An Architecture of VEI-Planners

Since the management of queries is quite different from the planning task, it is natural to consider a class of VEI-planners in which planning and query management are handled by two different processes. Therefore, we propose a class of VEI-planners as shown in Figure 3. In these VEI-planners, the planning task is conducted by a EI-planner $\hat{\mathcal{A}}$, and the queries and answers are managed by an event-driven procedure called the *VEI-controller*, whose functionality is encoded as a *reactive query policy*. $\hat{\mathcal{A}}$ does not interact with the external world directly; instead, all the requests for values made by $\hat{\mathcal{A}}$ are managed by the VEI-controller, which continuously waits for events such as requests for values by $\hat{\mathcal{A}}$ and answers from information sources. Once an event is received, the VEI-controller will choose one of the rules in the reactive query policy and "fire" it—generating a sequence of commands to control the planning process of $\hat{\mathcal{A}}$ and the issue of queries.

The VEI-controller controls the planning process of $\hat{\mathcal{A}}$ though the help of a data structure called *execution tree*, which allows the VEI-controller to store execution states of $\hat{\mathcal{A}}$ and then later on resume the execution of $\hat{\mathcal{A}}$ from one of the stored execution states. Likewise, the VEI-controller manages answers returned from information sources by a *buffer*, which holds the latest answers returned from information sources.

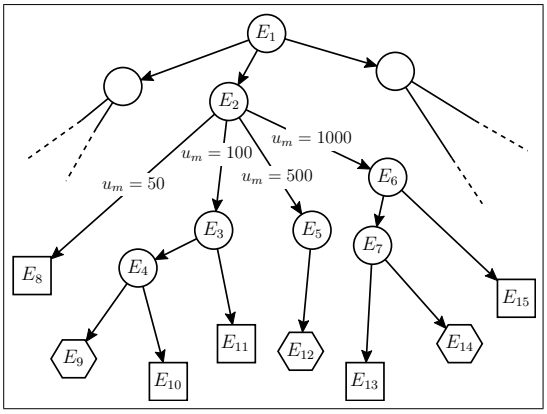

Fig. 4. An execution tree. Circles represent execution states where values have been requested. Edges represent substitutions for these values. For example, u_m is requested in state E_2, and the four edges below E_2 represent the four different possible values f u_m. Hexagons represent terminal execution states from which a solution can be extracted. Squares represent the latest non-terminal execution states of execution traces.

A. EI-planner and Execution Tree

Figure 4 illustrates the structure of an execution tree. Different paths from the root correspond to execution traces of $\hat{\mathcal{A}}$ with a different values for the unknowns. Each internal node corresponds to a *value-requesting execution state*, i.e., an execution state in which $\hat{\mathcal{A}}$ has requested a value for an unknown; and the edges emanating from this node represent different answers to the query (hence different values for the unknown). Each leaf node corresponds to the latest (terminal or non-terminal) execution state that $\hat{\mathcal{A}}$ has generated in this particular execution trace.

While $\hat{\mathcal{A}}$ is executing, the VEI-controller maintains a pointer to the *current node* in the execution tree, i.e., the node that corresponds to $\hat{\mathcal{A}}$'s current execution state. The VEI-controller does not update this pointer constantly during $\hat{\mathcal{A}}$'s execution, but only when the $\hat{\mathcal{A}}$ suspends itself.

The execution tree grows as value substitutions occur. When $\hat{\mathcal{A}}$ requests the value of some unknown u, it suspends itself, and the VEI-controller stores $\hat{\mathcal{A}}$'s execution state E at the current node. Later, the VEI-controller can substitute u for a value in E. When it does so, it will create a new child node E' for E that contains the execution state of $\hat{\mathcal{A}}$ right after the substitution in E'. It will label the edge between E and E' with the value substituted for u.

The set of commands that the VEI-controller can use to control the EI-planner $\hat{\mathcal{A}}$ and the execution tree are:

- Resume(E): resume $\hat{\mathcal{A}}$ from the execution state E. Preconditions: $\hat{\mathcal{A}}$ has been suspended. Effects: $\hat{\mathcal{A}}$ is resumed, and E is designated as the current node.
- Suspend: suspend the execution of $\hat{\mathcal{A}}$. Preconditions: $\hat{\mathcal{A}}$ is running. Effects: $\hat{\mathcal{A}}$ is suspended, and the current execution state of $\hat{\mathcal{A}}$ will be stored at the current node.
- Substitute(E, u, v): replace all the unknown symbol u in an execution state E with v. Preconditions: E is a value-requesting execution state. Effects: the execution state after the substituion will be stored as a new child node of E.

The EI-planner $\hat{\mathcal{A}}$ will generate the following event:

- Request(E, u): $\hat{\mathcal{A}}$ has requested a value for an unknown u at the execution state of the node E. Notice that when this event occurs, $\hat{\mathcal{A}}$ will automatically suspend itself.
- Terminate(E): $\hat{\mathcal{A}}$ has terminated at execution state E.

The VEI-controller can retrieve various information from the EI-planner and the execution tree using the following functions:

- Running(): return true iff the EI-planner is running.
- Suspended(): return true iff the EI-planner is suspended.
- Depended(E): return the set of all unknowns that are substituted for some values in some parent nodes of E in the execution tree. We call these unknowns the *depended unknowns* of the execution state E.
- MatchAnyNode(): return the execution state E in the execution tree such that the values of the depended unknowns of E are in the buffer. These values can be either valid or expired.
- MatchValidNode$_{\geq T}$(): return the execution state E such that the values of the depended unknowns of E will remain valid for at least a period of time T.

B. Buffer

The buffer stores the latest values of unknowns and the corresponding expiration time. Initially, the buffer is empty. When the VEI-planner receives an answer $ans(q) = (u, v(q), t_{expire}(q))$, the VEI-planner will automatically store $ans(u)$ in the buffer. If the buffer contains a previous answer of u, the new answer will replace the old one. In addition, there is a counter query_num(u) for each unknown u that records the number of unanswered queries. When the VEI-planner issues a query for u, it will increase query_num(u) by 1; when the VEI-planner receives an answer for u, it will decrease query_num(u) by 1.

There is no command for the VEI-controller to control the buffer, since the buffer is fully-automated. But there is one command that could affect the buffer indirectly:

- Issue(u): issue a query for an unknown u to the information source I_u. Preconditions: none. Effects: increase the counter query_num(u).

The buffer can generate these events:

- Answer(u, v, t_{expire}): the answer of a previously issued query for the unknown u has arrived; the answer is (u, v, t_{expire}).
- WillExpire$_{\leq T}$(u): the value of u in the buffer will expire after a period of time T.
- Expired(u): the value of u in the buffer has expired.

The VEI-controller can retrieve various information from the buffer by the following functions:

- ValueOf(u): return the (expired or valid) value of the unknown u in the buffer.
- NoQuery(u): return true iff query_num(u) $= 0$.
- SomeQuery(u): return true iff query_num(u) > 0.

```
Procedure VEI-controller(Ω)
  Input: A reactive query policy Ω
Loop
  Wait until the event queue is not empty
  Remove the first event e from the event queue
  Find a rule ((ε, φ) → λ) ∈ Ω, such that e and ε are
    unifiable and the condition φ[φ] is true, where
    φ is the substitution in the unification of e and ε
  Generate(λ[φ])
End Loop

Procedure Generate(λ)
  If λ is a primitive command, then
    Output c; If c is Return(E), then stop the VEI-planner.
  Else if λ is a list of commands (λ1, λ2, ..., λn), then
    Generate(λ1); Generate((λ2, ..., λn))
  Else if λ is a foreach statement (foreach x ∈ S, λ'), then
    For each y ∈ S, Generate(λ'[x\y])
  Else if λ is a if-then statement (if p then λ'), then
    Evaluate the condition p. If p is true, then Generate(λ')
```

Fig. 5. The VEI-controller. Ω is a reactive query policy.

- $\mathsf{Never}(u)$: return true iff the buffer has no answer for u;
- $\mathsf{Valid}_{\geq T}(u)$: return true iff the buffer has an answer for u that will remain valid for at least time T.
- $\mathsf{Valid}_{<T}(u)$: return true iff the buffer has an answer for u that will expire within time T.
- $\mathsf{Expired}(u)$: return true iff the answer for u has expired.

C. The VEI-controller and Reactive Query Policies

The VEI-controller is an event-driven procedure that generates commands according to a *reactive query policy*, a special type of reactive plan. A reactive query policy is a set of rules, each of them has the form $(\varepsilon, \varphi) \rightarrow \lambda$, where ε is an *event statement*, φ is a *condition statement*, and λ a *command statement* (We will define them below). A rule states that if the event that just occurs matches the event statement ε and the condition statement φ is true, then the VEI-controller should generate commands according to the command statement λ. Figure 5 shows the pseudo-code of the VEI-controller.

The inputs and outputs of VEI-controller is in the form of *message passing*. There are two types of messages: events and commands. The formal description of a message is something like $msg(v_1, v_2, ..., v_m)$, where msg is a message identifier and $v_1, v_2, \ldots, v_m$ are parameters ($m \geq 0$). In Section III-A and Section III-B, we have described the five events that can be generated by the EI-planner and the buffer: $\mathsf{Request}(E, u)$, $\mathsf{Terminate}(E)$, $\mathsf{Answer}(u, v, t_{expire})$, $\mathsf{WillExpire}_{\leq T}(u)$, and $\mathsf{Expired}(u)$. We have also described four commands: $\mathsf{Resume}(E)$, $\mathsf{Suspend}$, $\mathsf{Substitute}(E, u, v)$, and $\mathsf{Issue}(u)$. We call these commands the *primitive commands*. In addition to these primitive commands, there are two more primitive commands: (1) Nil, which means that the VEI-controller should do nothing, and (2) $\mathsf{Return}(E)$, which means that the VEI-controller should terminate the VEI-planner and return a solution in the terminal execution state E.

Events from different sources are temporarily stored in an *event queue*. Then the VEI-controller retrieves events one by one and processes them. There is no message queue for commands: commands, once generated, will immediately be executed. We assume that all commands can be carried out instantly, since the execution time of the commands are small when compared to the execution time of $\hat{\mathcal{A}}$ and the lag times of queries. Therefore there is no need to store commands in a queue.

A reactive query policy is a set of rules, each of them has the form $(\varepsilon, \varphi) \rightarrow \lambda$. The event statement ε of a rule is like event messages except that some of the parameters are variables. The free variables in φ and λ must be the variables in the event statement. When an event occurs, the VEI-controller will select a rule $(\varepsilon, \varphi) \rightarrow \lambda$ from the reactive query policy such that (1) ε is unificable with the event by a substition ϕ, and (2) $\varphi[\phi]$ (which is φ those free variables are substituted by values according to ϕ) is evaluated to be true.

For example, consider a reactive query policy that contains a rule $(\varepsilon, \varphi) \rightarrow \lambda$, where $\varepsilon = \mathsf{Request}(E, u)$, $\varphi = \mathsf{Valid}_{\geq T}(u)$, and $\lambda = \mathsf{Substitute}(E, u, \mathsf{ValueOf}(u))$. Note that E and u are free variables. Suppose an event $\mathsf{Request}(\mathsf{ES}_{13}, \mathsf{Price})$ occurs at time t (where ES_{13} denotes the 13'th execution state in the execution tree), and the current value for the unknown Price won't expire before time $t + T$. Then the event statement and the event are unificable by a substitution $\phi = [E \backslash \mathsf{ES}_{13}, u \backslash \mathsf{Price}]$, and the condition $\varphi[\phi] = \mathsf{Valid}_{\geq T}(\mathsf{Price})$ is true. Thus the VEI-controller should choose this rule and execute the command $\lambda[\phi] = \mathsf{Substitute}(\mathsf{ES}_{13}, \mathsf{Price}, \mathsf{ValueOf}(\mathsf{Price}))$—substitute symbols Price in ES_{13} with the current value of Price.

A condition statement of a rule is a first-order formula whose free variables are the variables in the event statement of the same rule. Each bound variable of a condition statement ranges over a set of unknowns, and each function symbol must be one of the following: Running, Suspended, NoQuery, SomeQuery, Never, $\mathsf{Valid}_{\geq T}$, $\mathsf{Valid}_{<T}$, Expired, ValueOf, MatchAnyNode, and $\mathsf{MatchValidNode}_{\geq T}$. The corresponding functions are described in Section III-A and Section III-B. The range of the bound variables is usually defined by the Depended function in Section III-A.

A command statement is defined recursively as follows: (1) a primitive command is a command statement; (2) a list of command statements is a command statement; (3) a foreach statement $(\mathsf{foreach}\ x \in S,\ \lambda)$, where λ is a command statement, x is a bound variable for λ, and S is the range of x, is a command statement; (4) an if-then statement $(\mathsf{if}\ p\ \mathsf{then}\ \lambda)$, where p is a condition statement and λ is a command statement, is a command statement. A command statement can be parsed by a recursive descent parser, as shown in the Generate function in Figure 5. During parsing, primitive commands are generated one by one.

Theorem 1 gives a sufficient condition for a reactive query policy Ω such that all solutions returned by Ω are T-valid solutions.

Theorem 1: A solution π returned by a reactive query policy Ω is T-valid if for every rule in Ω containing the command $\mathsf{Return}(\pi)$, the condition statement contains $(\forall u \in$

$\mathsf{Depended}(E), \mathsf{Valid}_{\geq T}(u))$.

Proof: The condition says that when $\tilde{A}$ terminates and return a solution π, all values of the depended unknowns of the terminal execution state will remain valid for at least a period of time T. By the definition of T-valid solutions, π is T-valid if this condition holds. ■

However, Theorem 1 is not a necessary condition. This means that it is possible to write a reactive query policy that always returns a T-valid solution, but the condition statements for some Return commands do not contain $\varphi' = (\forall u \in \mathsf{Depended}(E), \mathsf{Valid}_{\geq T}(u))$. But notice that there is no harm in appending φ' to any existing condition statement φ for Return commands (i.e., replacing every rule $(\varepsilon, \varphi) \rightarrow \lambda$ in which λ contains a Return command by $(\varepsilon, \varphi \wedge \varphi') \rightarrow \lambda$).

IV. A New Query Management Strategy

Both the eager strategy M_{eager} and the lazy strategy M_{lazy} in [8] can be formulated as reactive query policies. Since the set of reactive query policies is very large, there may be some that outperform the eager and lazy policies. One of them is our new *presumptive* query management strategy.

Like M_{eager}, the presumptive strategy issues queries for values when the old values expire. But unlike M_{eager}, it does not backtrack immediately, but continues as if the old value were still valid (like M_{lazy} does). Once the new value comes back from the information source, the presumptive strategy backtracks if the new value differs from the old one, and keeps going otherwise.

There are reasons to believe that in general, the presumptive strategy will perform better than both M_{eager} and M_{lazy}. First, when compared with M_{eager}, it does not wait for the information sources after a query is issued—instead, it does additional work that may save time later. Second, when compared with M_{lazy}, it issues queries more often, hence is more likely to backtrack from branches that may be incorrect.

The presumptive strategy can be written as a reactive query policy as follows.

Rule 1: $\mathsf{Request}(E, u) \wedge \mathsf{Never}(u) \rightarrow \mathsf{Issue}(u)$

Rule 2: $\mathsf{Request}(E, u) \wedge \mathsf{Valid}_{\geq T}(u) \rightarrow \mathsf{Substitute}(E, u, \mathsf{ValueOf}(u)), \mathsf{Resume}(\mathsf{MatchAnyNode}())$

Rule 3: $\mathsf{Request}(E, u) \wedge (\mathsf{Valid}_{<T}(u) \vee \mathsf{Expired}(u)) \rightarrow \mathsf{Substitute}(E, u, \mathsf{ValueOf}(u)), \mathsf{Resume}(\mathsf{MatchAnyNode}()), (\mathsf{if\ NoQuery}(u)\ \mathsf{then\ Issue}(u))$

Rule 4: $\mathsf{Answer}(u, v, t_{expire}) \wedge (\neg \mathsf{Valid}_{\geq T}(u)) \rightarrow \mathsf{Issue}(u)$

Rule 5: $\mathsf{Answer}(u, v, t_{expire}) \wedge (\mathsf{Valid}_{\geq T}(u) \wedge \mathsf{Running}()) \rightarrow \mathsf{Suspend}, \mathsf{Resume}(\mathsf{MatchAnyValue}())$

Rule 6: $\mathsf{Answer}(u, v, t_{expire}) \wedge (\mathsf{Valid}_{\geq T}(u) \wedge \mathsf{Suspended}()) \rightarrow \mathsf{Resume}(\mathsf{MatchAnyValue}())$

Rule 7: $\mathsf{WillExpire}_{\leq T} \wedge \mathsf{True} \rightarrow \mathsf{Issue}(u)$

Rule 8: $\mathsf{Terminate}(E) \wedge (\neg(\forall u \in \mathsf{Depended}(E), \mathsf{Valid}_{\geq T}(u))) \rightarrow \mathsf{Nil}$

Rule 9: $\mathsf{Terminate}(E) \wedge (\forall u \in \mathsf{Depended}(E), \mathsf{Valid}_{\geq T}(u)) \rightarrow \mathsf{Return}(E)$

Rule 10: $\mathsf{Expire}(u) \wedge \mathsf{True} \rightarrow \mathsf{Nil}$

Thus, it seems reasonable to expect that (1) the presumptive strategy will have a higher success rate than the other strategies, because it issues queries more often, and (2) in situations where all three strategies can terminate, the presumptive strategy will terminate more quickly. We now describe experimental tests of these hypotheses.

A. Experimental Evaluation

We wanted to compare the performance of the reactive query policies with planners that behaved very differently, and with different kinds of problem domains. For our planners, we used Graphplan [12] and SHOP2 [13]. For our problem domains, we used the well-known logistics domain, and the satellite domain from the AIPS-02 planning competition.

Figures 6–8 show our results for the logistics domain and satellite domain. Our tests were based on 16 logistics problems and 10 satellite problems chosen at random from the problem archives for the AIPS-2000 and AIPS-02 Planning Competitions. For each value of n (shown on the x axis in each figure) we did the following: we converted each of the logistics/satellite problems into a EI-problem by inserting n unknowns into the problem at random locations. For each of the EI-problems, we ran each combination of planner and reactive query policy 1000 times; thus each data point is the average of at least 10,000 trials. Within each trial, whenever the reactive query policy issued a query, we generated a value, a lag time, and a valid time at random, with the lag times and valid times varying from 1 to 10. The validity guarantee is 1. In each trial, we ran the reactive query policy until it either terminated or reached 100,000 seconds of CPU time.

Each figure contains nine lines, but some of them are so close as to be indistinguishable. For each of the three reactive query policies, there is a line for Graphplan in the logistics domain, a line for SHOP2 in the logistics domain, and a line for SHOP2 in the satellite domain.

Over all of the combinations of strategies, planners, and planning problems, there are 972,000 trials, with a total running time well in excess of 1×10^8 seconds—which hardly seems feasible for an experimental test! But there was an easy way to reduce the total amount amount of time needed for the test. For each combination of planner, strategy, and planning problem, we ran the planner on each path in the query tree, and kept track of the timing data for each edge of the tree. The timing data for each edge remains constant regardless of the query strategy, this enabled us to calculate the planner's running time for all of the trials very quickly.

The figures reveal several interesting things:

Success rates. As the number of unknowns increases, the eager strategy's success rate drops to 0. The eager strategy issues a query every time a value expires, and it waits to get the answer before proceeding; hence it easily ends up in situations where it fails because it never has a complete set of valid values at any one time. In contrast, both the presumptive and lazy strategies have success rates of 100%, except for the lazy strategy with 9 unknowns.

Average running times. On the semi-log plot used in the figure, the running times for the presumptive and lazy strategies are relatively straight except for small values of n, thus

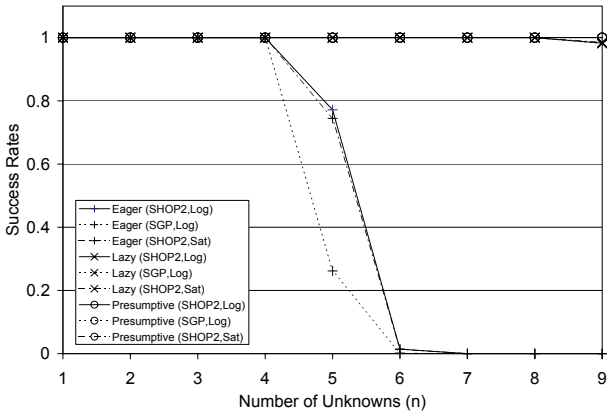

Fig. 6. Success rates for the three query strategies.

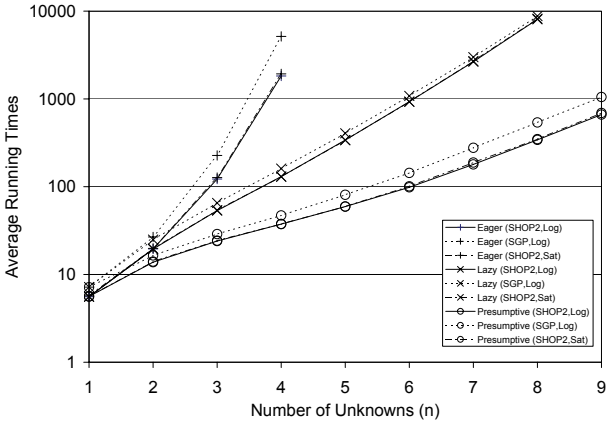

Fig. 7. Average CPU times in seconds.

both of them have exponential running time. The lines' slopes show that the lazy strategy's running time grew exponentially faster than the presumptive strategy's. At 8 unknowns, the presumptive strategy ran an average of 21.2 times as fast as the lazy strategy.

The data points for the eager strategy are missing for $n > 4$ because it failed to terminate within our time limit. The graph suggests that the running time for the eager strategy might be growing hyper-exponentially, but there are so few data points that it is difficult to tell.

Average number of queries. The results are similar to those for the running time. The lazy strategy's numbers grew exponentially faster the presumptive strategy's, and at $n = 8$ the lazy strategy made an average of 14.4 times as many queries as the presumptive strategy.

Query tree size. The query trees had 16 leaf nodes when $n = 4$, and 512 when $n = 9$. Thus, the lazy and presumptive strategies solved problems about 32 times as large as the eager strategy could solve.

Domain independence and planner independence. A result we hadn't expected was that the performance of each reactive

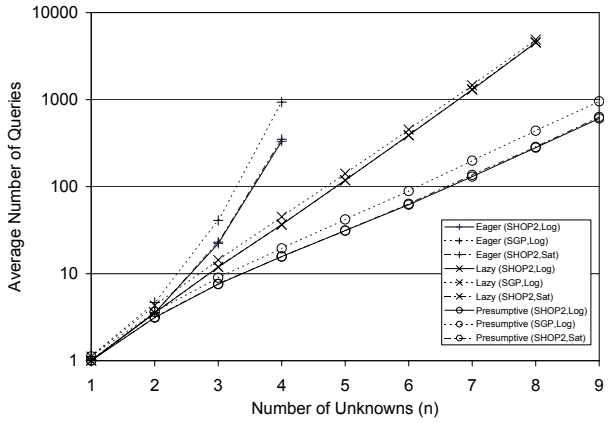

Fig. 8. Average number of queries.

query policy was almost completely independent of the planner and the planning domain. On every graph and for each strategy, the three lines for that strategy are nearly identical.

First, in both the logistics domain and the satellite domain, if we look at the original planning problems (without the unknowns), the problem is always solvable: there is always a path to the solution. For both planners, the amount of time needed to find this path was smaller than the overhead generated by the queries, so the time taken by the planner did not matter very much. What mattered most was how long it took to get valid values for all of the unknowns at the same time, and this was independent of the planner.

Second, in both problem domains, at each data point there were the same number of unknowns, and each unknown had roughly the same number of possible values, regardless of the domain. Thus, every query tree had roughly the same number of leaf nodes. In every path in the query tree, values were required for all of the unknowns, so every leaf node was at the same depth. Thus, all that mattered was how quickly a reactive query policy could get to that depth—which was independent of the problem domain.

Finally, each data point is the average of 1000 trials. Thus, the data points are not subject to the random variation that would occur in smaller experiments.

V. Related Work

The related areas of work include multi-agent environments where planners interact with other agents, and application areas such as multi-robot environments, distributed database management systems, servers distributed over the Internet, logistics, manufacturing, evacuation operations and games.

Our work differs from [8] in that our system guarantees a solution remain valid for some period of time after it is returned. This guarantee is important in mission-critical applications that cannot tolerate any fault in plan execution (e.g, robotic control in space exploration or factories). Such guarantees are impossible unless the expiration times (or at least a lower bound on it) are known.

As discussed in [8], our problem shares some aspects of the contingent planning problem with partial observability [7], [14], [15]. The key difference is that our sensing actions are queries that are executed during plan generation rather than during plan execution. The interleaving of planning and execution [7] is sometimes a way to deal with the expiration of values, but it relies on the detection of the failure during plan execution—and in some applications this information is not available, or failures must be avoided entirely.

A reactive query policy can be considered as a special type of reactive plan [16]–[19]. The management of the expiration of answers in real time shares some aspects of the works in real time searching [20] and real-time path planning [21], [22]. The adaption of new information makes use of the techniques in plan adaption [23]–[25], especially plan reuse [26], which is exactly how our query management strategy resume previously saved runtime stacks of A. The continuation of planning based on assumption making resembles PUCCINI [27], a partial-order planner that allows the option of assuming that certain preconditions hold, performing the action, and verifying the preconditions afterward.

VI. Conclusions

For integrating planning and data mining, an important problem is how the planner should deal with volatile external information. This paper has focused on how to do that.

We have introduced a general model for planning with volatile external information, including the notion of a T-valid plan that will remain valid for at least some time T after it is generated. We have developed a formalism for *reactive query policies* that can be used to adapt existing planners to deal with volatile information. We have stated the conditions under which the plan returned by a reactive query policy is T-valid. Finally, we have introduced a new query management policy called the *presumptive* strategy.

In our experiments, the presumptive strategy ran exponentially faster than the best previous strategy, the lazy strategy [8]. On the largest problems that we tried, the presumptive strategy took about 1/21 the time and generated about 1/14 as many queries as the lazy strategy The running times were relatively independent of both the problem domains and the planners, which suggests that the presumptive strategy may be better than the lazy strategy across many problem domains.

In some informal experiments that we have not reported in this paper, we have devised several other query management strategies and tested the presumptive strategy against them. In each case the presumptive strategy did best. We conjecture that under certain conditions the presumptive strategy is an optimal policy, i.e., that no other query management policy has a smaller average running time. In the future, we intend to do a mathematical analysis to confirm or deny this conjecture.

Acknowledgment

This work supported in part by ISLE contract 0508268818 (subcontract to DARPA's Transfer Learning program), NSF grant IIS0412812, and AFOSR grants FA95500510298 and Z802701. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

[1] W. H. McRaven, *Spec Ops : Case Studies in Special Operations Warfare: Theory and Practice*. Presidio Press, 1996.

[2] i2 Technologies, "Reducing planning cycle time at altera corporation," http://www.i2.com/assets/pdf/96FDF2C7-71C7-43B5-906A01BAE2F0AE76.pdf, 2002.

[3] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler, "Information gathering during planning for web services composition," in *ISWC*, 2004, pp. 335–349.

[4] ——, "Information gathering during planning for web services composition," *Journal of Web Semantics*, vol. 3, no. 2–3, pp. 183–205, 2005.

[5] A. Depoutovitch and A. Wainstein, "Building grid-enabled data-mining applications," *Dr. Dobb's Journal*, 2005.

[6] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," http://www.globus.org/alliance/publications/papers/ogsa.pdf, 2002.

[7] C. A. Knoblock, "Planning, executing, sensing, and replanning for information gathering," in *IJCAI*, 1995, pp. 1686–1693.

[8] T.-C. Au, D. Nau, and V. Subrahmanian, "Utilizing volatile external information during planning," in *ECAI*, 2004, pp. 647–651.

[9] T.-C. Au and D. Nau, "The incompleteness of planning with volatile external information," in *Proceedings of the European Conference on Artificial Intelligence*, 2006, pp. 839–840.

[10] M. J. Fischer and N. A. Lynch, "Impossibility of distributed consensus with one faulty," *JACM*, vol. 32, no. 2, pp. 374–382, 1985.

[11] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.

[12] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," in *IJCAI*, 1995, pp. 1635–1642.

[13] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *JAIR*, vol. 20, pp. 379–404, December 2003, http://www.jair.org/abstracts/nau03a.html.

[14] D. Draper, S. Hanks, and D. Weld, "Probabilistic planning with information gathering and contingent execution," in *AIPS*, 1994, pp. 31–36.

[15] M. A. Peot and D. E. Smith, "Conditional nonlinear planning," in *AIPS*, 1992, pp. 189–197.

[16] M. Beetz and D. McDermott, "Declarative Goals in Reactive Plans," in *AIPS*, 1992.

[17] M. Drummond, "Situated control rules," in *KR*, 1989, pp. 103–113.

[18] J. R. Firby, "An investigation into reactive planning in complex domains," *Artif. Intel.*, vol. 3, pp. 251–288, 1987.

[19] M. Schoppers, "Universal plans for reactive robots in unpredictable environments," in *IJCAI*, 1987, pp. 1039–1046.

[20] R. E. Korf, "Real-time heuristic search," *Artif. Intel.*, vol. 42, no. 2–3, pp. 189–211, 1990.

[21] S. Koenig and R. G. Simmons, "Solving robot navigation problems with initial pose uncertainty using real-time heuristic search," in *AIPS*, 1998, pp. 145–153.

[22] A. Stentz, "Optimal and efficient path planning for partially-known environment," in *IEEE International Conference on Robotics and Automation*, 1994, pp. 3310–3317.

[23] K. J. Hammond, *Case-Based Planning: viewing learning as a memory task*. Academic Press, 1989.

[24] S. Hanks and D. S. Weld, "A domain-independent algorithm for plan adaptation," *JAIR*, vol. 2, pp. 319–360, 1995.

[25] S. Kambhampati and J. A. Hendler, "A validation-structure-based theory of plan modification and reuse," *Artif. Intel.*, vol. 55, no. 2–3, pp. 193–258, 1992.

[26] L. Ihrig and S. Kambhampati, "Plan-space vs. state-space planning in reuse and replay," Department of Computer Science, Arizona State University, Tech. Rep. 94-006, 1996.

[27] K. Golden, "Leap before you look: Information gathering in the puccini planner," in *AIPS*, 1998, pp. 70–77.