Information Gathering During Planning for Web Service Composition [⋆]

Ugur Kuter [a] , Evren Sirin [a] , Bijan Parsia [b] , Dana Nau [a], James Hendler [a]

[a] *University of Maryland, Computer Science Department, College Park MD 20742, USA*

[b] *University of Maryland, MIND Lab, 8400 Baltimore Ave, College Park MD 20742, USA*

Abstract

Hierarchical Task-Network (HTN) based planning techniques have been applied to the problem of composing Web Services, especially when described using the OWL-S service ontologies. Many of the existing Web Services are either exclusively information providing or crucially depend on information-providing services. Thus, many interesting service compositions involve collecting information either during execution or during the composition process itself. In this paper, we focus on the latter issue. In particular, we present ENQUIRER, an HTN-planning algorithm designed for planning domains in which the information about the initial state of the world may not be complete, but it is discoverable through plan-time information-gathering queries. We have shown that ENQUIRER is sound and complete, and derived several mathematical relationships among the amount of available information, the likelihood of the planner finding a plan, and the quality of the plan found. We have performed experimental tests that confirmed our theoretical results and that demonstrated how ENQUIRER can be used for Web Service composition.

Key words: HTN Planning, Information Gathering, Web Service Composition

⋆ This is a revised and extended version of a paper that was presented at ISWC-2004 with the same title in November 2004.

Email addresses: `ukuter@cs.umd.edu` (Ugur Kuter), `evren@cs.umd.edu` (Evren Sirin), `bparsia@isr.umd.edu` (Bijan Parsia), `nau@cs.umd.edu` (Dana Nau), `hendler@cs.umd.edu` (James Hendler).

1 Introduction

Web Services are Web accessible, loosely coupled chunks of functionality with an interface described in a machine readable format. Web Services are designed to be *composed*, that is, combined in workflows of varying complexity to provide functionality that none of the component services could provide alone. AI planning techniques can be used to automate Web Service composition by representing services as actions, and treating service composition as a planning problem. On this model, a service composition is a ground sequence of service invocations that accomplishes a goal or task.

OWL-S [1] provides a set of ontologies to describe Web Services in a more expressive way than allowed by the Web Service Description Language (WSDL). In OWL-S, services can be described as complex or atomic processes with preconditions and effects. This view enables us to translate the process-model constructs directly to HTN methods and operators [2]. Thus, it is possible to use HTN planners to plan for composing Web Services that are described in OWL-S.

Using AI planning techniques for Web Services composition introduces some challenges. In particular, traditional planning systems assume that the planner begins with complete information about the world. However, in service composition problems, most of the information (if it is available at all) must be acquired from Web Services, which may work by exposing databases, or may require prior use of such information-providing services. For example, consider the scenario described in the Scientific American article about the Semantic Web [3]. This scenario involves two people who are trying to take their mother to a physician for a series of treatments and follow-up meetings. The planning problem in this scenario is to come up with a sequence of appointments that will fit in to everyone's schedules, and at the same time, to satisfy everybody's preferences. In this setting, the planner needs to determine an available appointment time first, or to learn the locations of the treatment centers. In many cases, however, it is not feasible or practical to execute all the information-providing services up front to form a complete initial state of the world. In such cases, it makes sense to gather the required information during planning.

In this paper, we describe ENQUIRER, an HTN-planning algorithm that can solve Web Service composition problems that require gathering information during the composition process. ENQUIRER is based on the SHOP2 planning system [4], and designed for planning domains in which the information about the initial world state may not be complete. In such cases, ENQUIRER issues queries to obtain the necessary information, it postpones all decisions related to that information until a response comes in, and it continues examining alternative branches of the search space. By gathering extra information at plan time, the planner is able to explore many more branches in the search space than the initial state ordinarily permits. Since external queries often dominate planning time, and, being distributed, are strongly parallelizable, ENQUIRER's non-blocking strategy is sometimes able to dramatically improve the time to find a plan. We investigate two different extensions to this base strategy: (1) Continue planning in a search branch as soon as one answer is available from any information source and use the information from other sources only when a plan (or a plan of sufficient quality) cannot be found. (2) Handle conflicting information coming from different sources when there is an explicit ranking criteria that tells how much an information source should be trusted.

We also provide the sufficient conditions to ensure the soundness and completeness of ENQUIRER, derive a recurrence relation for the probability of ENQUIRER finding a plan, and prove theoretical results that give mathematical relationships among the amount of information available to ENQUIRER and the probability of finding a plan. These relationships are confirmed by our experimental evaluations. We describe how the generic ENQUIRER algorithm can be used to solve the problem of composing Web Services, and test the efficiency of the algorithm on two real-world problems.

2 Motivations

HTN-planning algorithms have proven promising for Web Service composition. Many service-oriented objectives can be naturally described with a hierarchical structure. HTN-style domains fit in well with the loosely-coupled nature of Web Services: different decompositions of a task are independent so the designer of a method does not have to have close knowledge of how the further decompositions will go. Hierarchical modeling is the core of the OWL-S [1] process model to the point where the OWL-S process model constructs can be directly mapped to HTN methods and operators. In our previous work [2], we have shown how such a translation can be done for SHOP2 [4]. In this work, we have kept the basic SHOP2-language mapping intact, and focused on extending the way SHOP2 deals with plan-time information gathering. [1] We call the extended algorithm ENQUIRER.

We have identified three key features of service-oriented planning:

[1] In this paper, we focus on information gathering as plan-time execution of Web Services. Nothing in this work, however, is specific to information-providing Web Services, and could be immediately adapted to any oracular query-answering mechanism, e.g., a user could interactively supply answers to the system.

- *The planner's initial information about the world is incomplete.* When the size and nature of Web is considered, we cannot assume the planner will have gathered all the information needed to find a plan. As the set of operators and methods grows very large (i.e., as we start using large repositories of heterogeneous services) it is likely that trying to complete the initial state will be wasteful at best and practically impossible in the common case.
- *The planning system should gather information during planning.* While not all the information relevant to a problem may have already been gathered, it will often be the case that it is accessible to the system. The relevance of possible information can be determined by the possible plans the planner is considering, so it makes sense to gather that information while planning.
- *Web Services may not return needed information quickly, or at all.* Executing Web Services to get the information will typically take longer time than the planner would spend to generate plans. In some cases, it will not be known a priori which Web Service gives the necessary information and it will be required to search a Web Service repository to find such capable services. It may not be possible at all to find those services. Furthermore, in some cases the found service cannot be executed because the service requires some password that the user cannot provide or the service is inaccessible due to some network failure. The system should not cease planning while waiting for answers to its queries, but keep planning to look for other plans that do not depend on answering those specific queries.

ENQUIRER is designed to address all of the issues above. In the subsequent sections, we first give a brief background on HTN Planning and the SHOP2 planning system, and then we present the ENQUIRER planning algorithm, as well as our theoretical and experimental evaluations of it.

3 Background: HTN Planning and SHOP2

The purpose of an HTN planner is to produce a sequence of actions that perform some activity or *task*. The description of a planning domain includes a set of planning *operators* and *methods*, each of which is a prescription for how to decompose a task into its *subtasks* (smaller tasks). The description of a planning problem contains an initial state as in classical planning. Instead of a goal formula, however, there is a partially-ordered set of tasks to accomplish.

Planning proceeds by decomposing tasks recursively into smaller and smaller subtasks, until *primitive tasks*, which can be performed directly using the planning operators, are reached. For nonprimitive each task, the planner chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks or the interactions among them prevent the plan from being feasible, the planning system backtracks and tries other methods.

SHOP2 is an HTN planner that generates actions in the order they will be executed in the world. Its backtracking search considers the methods applicable to the same task in the order they are specified in the knowledge base given to the planner. In SHOP2, for each primitive task, the description of a planning domain must include only one planning operator. For nonprimitive tasks, on the other hand, there can be one or more methods in a domain description. As an example, consider a Delivery Domain, in which the task is to deliver a box from one location to another. Figure 1(a) shows two SHOP2 methods for this task: *delivering by car*, and *delivering by truck*. Delivering by car involves the subtasks of loading the box to the car, driving the car to the destination location, and unloading the box at the destination. Note that each method's preconditions are used to determine whether or not the method is applicable: thus in Figure 1(a), the *deliver by car* method is only applicable if the delivery is to be a fast one, and the *deliver by truck* method is only applicable if it can be a slow one.

Now, consider the task of delivering a box from the University of Maryland to MIT and suppose we do not care about a fast delivery. Then, the *deliver by car* method is not applicable, and we choose the *deliver by truck* method. As shown in Figure 1(b), this decomposes the task into the following subtasks: *(1)* reserve a truck from the delivery center at Laurel, Maryland to the center at Cambridge, Massachusetts, *(2)* deliver the box from the University of Maryland to Laurel, *(3)* drive the truck from Laurel to Cambridge, and *(4)* deliver the box from Cambridge to MIT. The tasks for reserving a truck and for driving the truck are primitive and they are accomplished by the planning operators specified for them. For the two delivery subtasks produced by this decomposition, we must again consider our delivery methods for further decomposing them until we do not have any other task to decompose.

As any traditional HTN planner, SHOP2 evaluates the preconditions of the operators and methods with respect to the world state it maintains locally during planning and it assumes that it has all the required information in its local state in order to evaluate these preconditions. For example, in our delivery example, it is assumed that the planner knows all the routes for traveling by car between the initial and final locations so that it can determine which route to take. Certainly, this information may not be available before the planning process starts. Considering that the amount of information available on the Web is huge, a planner should gather the information as needed by the planning process. Since gathering information may take some considerable amount of time, it would be wise to continue planning while the queries are being processed. For example, the planner can continue planning with the *truck delivery* method until the answer to the query about the possible routes

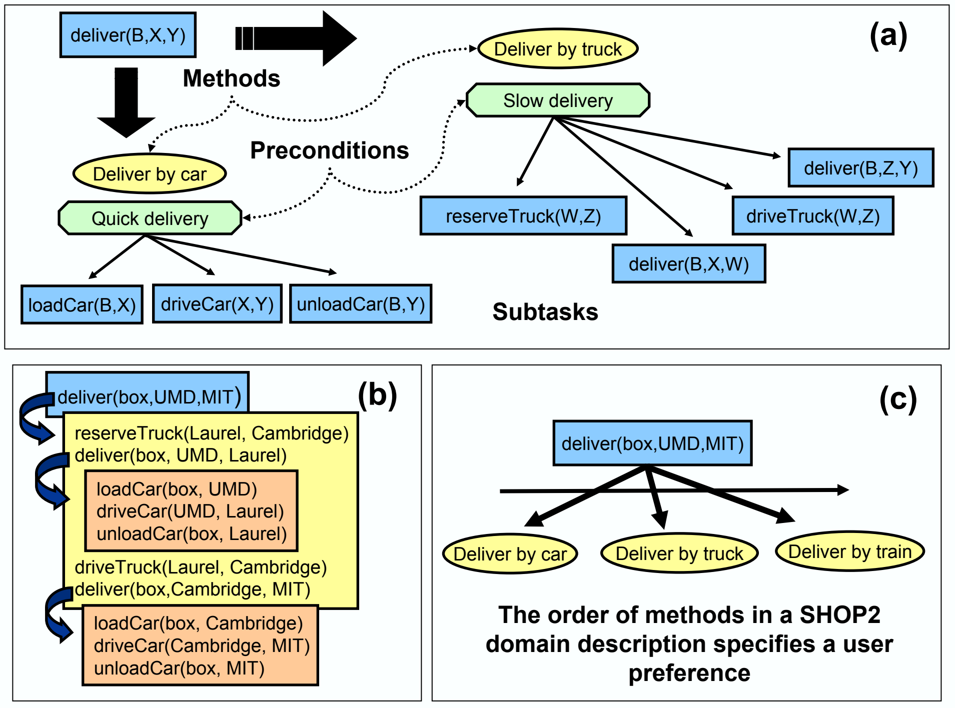

Fig. 1. Delivery planning example.

has been received.

As described above, SHOP2 considers the methods applicable to a nonprimitive task in the order they are specified in the knowledge base given to the planner. This feature of the planner allows for specifying *user preferences* among such methods, and therefore, among the solutions that can be generated using those methods. For example, in our delivery example, suppose we have the alternative of delivering the box by train, in addition to the *delivery by car* and *deliver by truck* methods. Furthermore, suppose delivering the box by car is the fastest delivery method among the three available alternatives, and delivering by train is the slowest one. Then, we can specify a user preference for fastest possible delivery method by ordering these three methods in SHOP2's knowledge base, as illustrated in Figure 1(c).

4 Definitions and Notation

We use the same definitions for variables, constants (i.e., objects), logical atoms, states, task symbols, tasks, task networks, actions, operators, methods, and plans as in SHOP2. A state is *complete* if it contains all of the atoms that are true in the world. Otherwise, it is *incomplete*. A *plan* is a sequence of actions (i.e., ground operator instances). As in SHOP2, we denote a plan as an ordered set $\{o_1, o_2, \ldots, o_k\}$, where each o_i is a ground instance of an operator.

A *query* is an expression of the form

$$Q(\vec{x}) \leftarrow C(\vec{y}),$$

where Q is the unique label of the query, C is a conjunction of (possibly unground) literals, $\vec{y}$ is the set of variables appearing in C and $\vec{x}$ is the set of variables whose values are being sought. In this paper, we require that $\vec{x} = \vec{y}$. If $\vec{x} = \vec{y} = \emptyset$ then Q is said to be ground.

The intent of a query is to gather information during planning about the state of the world before planning started (i.e., about the initial state). For example,

Example 1 (Query) *Suppose we want to find the hospitals located in Maryland that provides eye treatment services. Also suppose we have types* $City$, $State$, $Hospital$ *and* $Treatment$ *along with the relations* $hasCity(State, City)$, $locatedIn(Hospital, City)$ *and* $provides(Hospital, Treatment)$. *Then the corresponding query can be written as*

$$\begin{aligned} Q(H, C) \leftarrow\, & Hospital(H) \wedge locatedIn(H, C) \wedge City(C) \wedge \\ & hasCity(\mathbf{maryland}, C) \wedge provides(H, \mathbf{eye\_treatment}), \end{aligned}$$

where **maryland** *and* **eye_treatment** *are objects, and* H *and* C *are variables.*

The *answer* for a query $Q(\vec{x}) \leftarrow C(\vec{y})$ in a state S is a set of variable substitutions $A = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, where σ_k specifies a value for each variable in $\vec{x}$ such that all the positive atoms in $\sigma_k(C)$ are in state S and none of the negative atoms in $\sigma_k(C)$ are in S.

An *information source* is defined to be any external source that can provide information during planning. The specification of an information source describes the contents of the information provided. More formally, the specification of an information source is

$$I(\vec{x_i}; \vec{x_o}) \rightarrow D(\vec{y}),$$

where I is the unique label of the source, D, called the *body* of the source description, is a conjunction of logical atoms, $\vec{y}$ is the set of variables appearing in D, $\vec{x_i}$ is a set of input variables, and $\vec{x_o}$ is a set of output variables. An information source is *fully specified* if we have $\vec{x_i} \cup \vec{x_o} = \vec{y}$. In this paper, we require that the information-source specifications to be fully specified.

Example 2 (Information Source) *Suppose that there is an information source that returns all the cities located in a state for any given state. The specification of this source would be:*

$$I(S; C) \rightarrow State(S) \wedge hasCity(S, C) \wedge City(C).$$

The reason we require that information sources be fully specified is that the answer for a query is a variable substitution that defines a value for each variable in the head of that query. That variable substitution is applied to the body of the source description and the ground predicates can be inserted in the state of the world. If the description of an information source contains existentially-quantified variables (i.e., variables that do not appear in the list of input or output variables) then we need to introduce new objects represented by skolem constants, which would make the reasoning process more complex.

An information source I can be queried by supplying a variable substitution Θ that specifies a value for each of the input variables in $\vec{x_i}$. The *result* of the execution of the information source is a set of variable substitutions $R(I, \Theta) = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ where each σ_k is a variable substitution that specifies a value for each variable in $\vec{x_o}$ such that $\sigma_k(\Theta(D))$ is true in the world. This means that the information returned is *sound* but does not need to be *complete* — i.e., $R(I, \Theta)$ may not contain all of the possible substitutions that make $\Theta(D)$ true.

Let X be a set of information sources. Then, we define $\delta(X)$ as the total amount of information that can be gathered from these sources. More formally,

$$\delta(X) = \bigcup_{I \in X, \sigma \in R(I,\Theta)} \sigma(\Theta(D)),$$

for every variable substitution Θ that specifies a value for every input variable in I.

A *complete-information planning problem* is a tuple $P^C = (S, T, D)$, where S is a complete initial state, T is a task network, and D is an HTN-domain description that consists of a set of planning operators O and methods M, respectively. A solution for the planning problem P^C is a sequence of actions (i.e., ground operator instances) that, when executed in the initial state, accomplishes the goal task network T.

A *search tree* $\mathcal{T}$ for a complete-information planning problem is defined recursively as follows:

- The root node is a pair (T, π) where T is the goal task network to be accomplished and π is the empty plan.
- Let (T', π') be a node in $\mathcal{T}$, and let t be a task that has no predecessors in T'. If t is primitive then let o be the planning operator for t. Then, the node (T'', π'') is the only child of (T', π') where T'' is the task network generated by removing t from T' and $\pi'' = \mathit{append}(\pi', o)$. We also label the arc from (T', π') to (T'', π'') with o.

  If t is not primitive then let m be a method instance applicable to t. Then, the node (T'', π') is a child of (T', π') where T'' is the task network

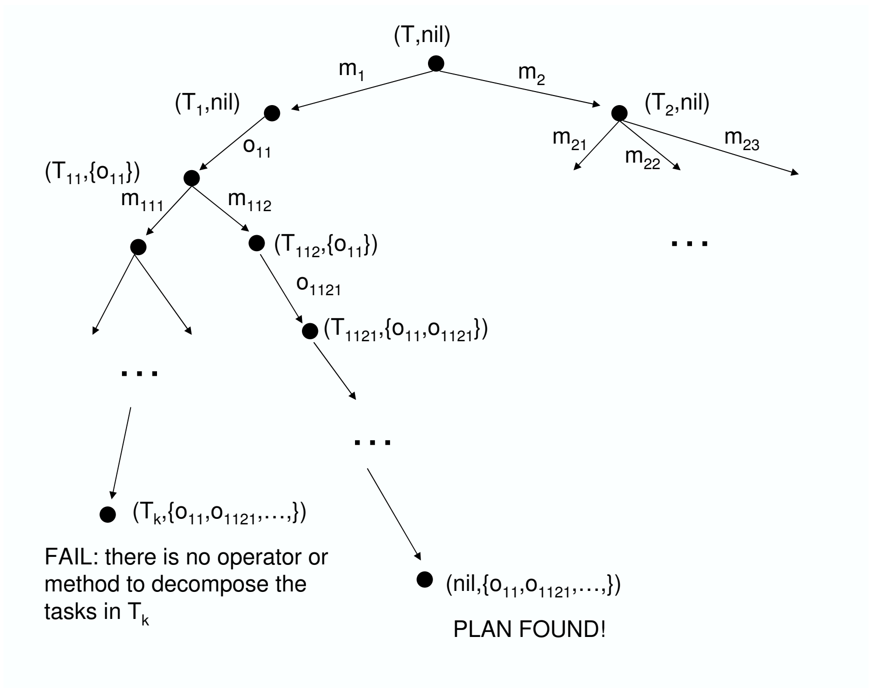

Fig. 2. The search tree for a complete-information planning problem.

generated from removing t from T' and adding the subtasks specified by m, preserving the ordering constraints among those tasks. We label the arc from (T', π') to (T'', π') with m.

Figure 2 shows an example of a search tree for a complete-information planning problem P^C. Note that the state information regarding a node $n = (T, \pi)$ in the search $\mathcal{T}$ are implicitly specified by the initial state S and the current partial plan π; that is, the state at each node in a search tree can be generated by simulating the effects of the actions in π in the initial state S. A node n in $\mathcal{T}$ is called a *terminal node* if n does not have any children in $\mathcal{T}$. A *search trace* in $\mathcal{T}$ of a complete-information planning problem P^C is the sequence of method and operator instances that label the arcs along a branch of $\mathcal{T}$. A search trace corresponds to a solution for P^C if the terminal node of that trace is of the form (nil, π); i.e., if it reaches to the empty task network which marks the successful termination of the planning process. Note that π is equal to the ground instances of the operators specified in the sequence of labels of such a search trace, which constitutes a solution for P^C.

An *incomplete-information planning problem* is a tuple $P^I = (J, X, T, D)$, where J is a set of ground atoms that are initially known, X is a set of information sources, T is a task network, and D is an HTN-domain description. The search tree for an incomplete-information problem is defined similarly to that of a complete-information problem, except that, in this case, the information available about the world in a node $n = (T, \pi)$ of the search tree is defined by T, π and the total information available about the initial state

$J \cup \delta(X)$ where J denotes an incomplete state of the world, and X is the set of available information sources.

An incomplete-information planning problem $P^I = (J, X, T, D)$ is *consistent* with a complete-information planning problem $P^C = (S, T, D)$ if and only if $J \cup \delta(X) \subseteq S$. We are assuming that the results returned from information sources will not change during the course of planning. In other words, for an information source I, $R(I, \sigma)$ is fixed and does not change during planning, and therefore, $\delta(X)$ is fixed — although, we may not know $\delta(X)$ a priori.

Note that the above restriction is the same *information persistence* assumption that is first introduced in [5]. However, we do not require that the information gathered from external sources cannot be changed by the planner itself as in [5]. This assumption specifies a class of service-composition problems in which the information collected from the Web Services are static during the planning process. Some examples of such service-composition problems is discussed in Section 9, where we present the experimental evaluation of our approach.

We define an *information-providing Web Service* as a service that returns information only about the initial state, and does not have any world-altering effects. It is straightforward to create an *information source specification* for an information-providing Web Service provided that the inputs, outputs, preconditions and effects of the service is known.

Example 3 (Information-providing OWL-S Service) *The OWL-S description (in OWL-S abstract syntax) that results in the information source specification of Example 2 is as follows:*

```
define atomic process FindCities(
    inputs: (s - State),
    outputs: (c - City),
    precondition: (),
    result: (hasCity(s, c)))
```

We define a *service-composition problem* to be a tuple of the form $P^W = (J, W, C, K)$, where J is a (possibly incomplete) initial state, W is a set of information-providing Web Services that are available during the planning process, C is a (possibly) composite OWL-S process, and K is a collection of OWL-S process models such that we have $C \in K$. A solution for P^W is a *service composition*, which is a sequence of atomic OWL-S processes such that, when executed, achieves the functionality desired by the process C.

Let $P^W = (J, W, C, K)$ be a service-composition problem. Then, we say that W is *equivalent* to an incomplete-information problem $P^I = (J, X, T, D)$, where T is the SHOP2 translation for the OWL-S process C, $D =$ TRANSLATE-PROCESS-MODEL(K) is the HTN-domain description generated by the translation algorithm of [6], and X is the information source specifications for services in W generated as explained above – i.e., we have $\delta(X) = \delta(W)$. Note that, we do not need the TRANSLATE-ATOMIC-PROCESS-OUTPUT function anymore because there is no need to encode where and when the information-providing services will be called. ENQUIRER uses the information source specifications to decide which service may provide relevant information during planning.

5 The ENQUIRER Planning Algorithm

ENQUIRER starts with an incomplete initial world state, gathers relevant information during planning, and continues to explore alternative possible plans while waiting for information to come in. The algorithm is shown in Fig.3. The input is an incomplete-information planning problem (J, X, T, D) as defined above. With this input, ENQUIRER first initializes the Query Manager by using the routine QueryManager::Initialize. We discuss the details of the Query Manager in the next section, so we defer the definition of this routine until then. Secondly, ENQUIRER initializes its $OPEN$ list to (T, π) where T is the goal task list, and π is the empty plan. Intuitively, the $OPEN$ list holds the information about the partial plans ENQUIRER has generated during planning and ENQUIRER may choose to explore any of these possibilities as explained next.

At each iteration of the planning process, we first check if the $OPEN$ list is empty. If so, then we report $failure$ since this means that every trace in the search tree for the input problem is exhausted without success. Otherwise, we select a tuple (T, π) from $OPEN$. Note that this selection is arbitrary in the pseudocode of Figure 3. However, in the implementation of the planner, we can exploit different selection mechanisms, which determines the way ENQUIRER performs its search under the conditions of incomplete information. For example, always selecting the first tuple in $OPEN$ implements a depth-first search strategy for ENQUIRER. We discuss the importance of different such implementations of search strategies in ENQUIRER in Section 9, where we discuss our implementation of the planner and experiments.

After selecting a tuple (T, π) from $OPEN$, the next step is to check if the task network T is empty or not. If so, we have π as our plan since all of the goal tasks have been accomplished successfully. Otherwise, we nondeterministically choose a task t in T that has no predecessors in T. If t is primitive then we decompose it using the operators in O. Otherwise, the methods in M must be used.

Suppose t is primitive. Then, we first find the operator o that matches to the

```
procedure ENQUIRER($J, X, T, D = (O, M)$)
 $\pi \leftarrow \emptyset$; $OPEN \leftarrow \{(T, \pi)\}$
 QueryManager::Initialize($J, X, T, \pi$)
 loop
  if $OPEN = \emptyset$ then return($failure$)
  select a tuple $(T, \pi)$ from $OPEN$ and remove it
  if $T = \emptyset$ then return($\pi$)
  nondeterministically choose a task $t \in T$ that has no predecessors
  if $t$ is a primitive task then
    let $o$ be an operator for $t$
    if there is no such operator in $D$ for $t$ then return($failure$)
    let $C$ be the preconditions of $o$
    $answer \leftarrow$ QueryManager :: IssueQuery($C, T, \pi$)
    $OPEN \leftarrow$ ProcessQueryResponse($t, o, T, \pi, OPEN, answer$)
  else
    let $m$ be a method for $t$
    if there is no such method in $D$ for $t$ then return($failure$)
    let $C$ be the preconditions of $m$
    $answer \leftarrow$ QueryManager :: IssueQuery($C, T, \pi$)
    $OPEN \leftarrow$ ProcessQueryResponse($t, m, T, \pi, OPEN, answer$)
```

Fig. 3. The ENQUIRER algorithm.

task t. If there is no such operator in O, then this is a backtracking point in the search; that is, ENQUIRER does not report failure for the input planning problem, but it backtracks and tries other tuples from the $OPEN$ list. Otherwise, ENQUIRER calls the Query Manager to gather information on the preconditions C of o. It does so by invoking the procedure QueryManager::IssueQuery as shown in Figure 3. If t is not primitive, then ENQUIRER finds a method m for the task t, and invokes QueryManager::IssueQuery to gather information on the preconditions C of m.

The QueryManager::IssueQuery procedure returns one of the following three types of $answer$s for a query request. First, it may return the empty set, which means that the precondition C cannot be satisfied in the world, given the incomplete information about the initial state of the world and the set of available information sources that can be queried. Secondly, QueryManager::IssueQuery may return $wait$, which means that the Query Manager have invoked one or more services to obtain information about C, but those services did not complete their executions. In this case, ENQUIRER proceeds with planning for other solutions by considering alternative search traces in its search space. Finally, the procedure may return a set of variable bindings $\{\sigma_i\}_{i=1}^{k}$ such that each variable binding σ_i can be used to ground the particular method or the operator and generate the successor search node in ENQUIRER. The ProcessQueryResponse is responsible for checking the answers from the Query Manager, process them, and update the $OPEN$ list of

```
procedure ProcessQueryResponse($t, x, T, \pi, OPEN, answer$)
 if $answer = \emptyset$ then return($\emptyset$)
 if $answer = wait$ then return($OPEN \cup (T, \pi)$)
 // Otherwise, $answer$ is a set of variable substitutions
 choose a substitution $\sigma \in answer$
 if $x$ is an operator then
   let $T'$ be the task network generated by removing $t$ from $T'$
   QueryManager::UpdateInternalState($\pi, \sigma(x)$)
   insert the tuple $(T', append(\pi, \sigma(x)))$ into the $OPEN$ list
 else
   let $T'$ be the task network generated by removing $t$ from $T'$
      and adding the subtasks specified by $\sigma(x)$
   insert the tuple $(T', \pi)$ into the $OPEN$ list
 return $OPEN$
```

Fig. 4. The ProcessQueryResponse procedure.

the planner accordingly.

The ProcessQueryResponse procedure takes as input a tuple of the form $(t, x, T, \pi, OPEN, answer)$, where t is the current task selected by ENQUIRER for decomposition, x is either an operator or a method depending on whether t is primitive or not, T is the current task network, π is the current partial plan, $OPEN$ is the current open list, and $answer$ is the answer received from the Query Manager. It first checks if the answer marks a failure in planning. If so, then it returns $\emptyset$. This way, the $OPEN$ list is set to be the empty list, and ENQUIRER returns $failure$ immediately in the next iteration.

If the $answer$ is not the empty set, then the next step is to check whether it is $wait$. This case occurs when the Query Manager starts to execute some Web Services to generate the answer for the current query, but those services has not finished their execution. Therefore, the planner must defer its decision about the information being queried, and possibly try other alternative search traces to find a solution. To do this, ProcessQueryResponse re-inserts the tuple (T, π) into $OPEN$ list and returns the updated $OPEN$ list. Note that the way the tuple (T, π) is re-inserted in to the $OPEN$ list depends on the way ENQUIRER selects tuples from this list during planning. As described earlier, ENQUIRER can exploit different selection mechanisma to implement different search methodologies. For example, if ENQUIRER always selects the first tuple in the $OPEN$, implementing a depth-first search, and if ProcessQueryResponse re-inserts tuples in the front of $OPEN$, then this will yield to implementing a busy waiting scheme, and in effect, to blocking the search until the Query Manager returns some answers for the current query. On the other hand, if we re-insert the current tuple (T, π) at the end of the $OPEN$ list, this will enable ENQUIRER to search alternative branches in the search space, and come back to the branch specified by (T, π) only when it cannot find any solutions in the alternative search branches.

If *answer* is not *wait*, then it is a set of variable substitutions for the variables that appear in the head of the query that was issued. In this case, ProcessQueryResponse nondeterministically chooses one of such substitutions σ, and applies it to the operator or the method specified by x in its input. Then, it generates the successor task network to be accomplished as follows. If t is primitive and x is an operator then ProcessQueryResponse first generates the ground operator instance, i.e., the action, $\sigma(x)$. Then, it removes t from T, sends the changes that needs to be made in the state of the world due to the effects of this action the Query Manager, and updates the current partial policy with the action $\sigma(x)$. If t is not primitive and x is method for it, then ProcessQueryResponse first generates the ground method instance, $\sigma(x)$. Then, it updates the current task network with the subtasks in $\sigma(x)$. Finally, in either of the cases above, ProcessQueryResponse updates the $OPEN$ list with the newly-generated task network and policy, and returns it to ENQUIRER.

6 The Query Manager

The Query Manager is responsible for processing and directing ENQUIRER's queries about the world to the available Web Services, collecting the information gathered from them, processing this information for consistency and correctness of the planning process, and returning the information to ENQUIRER to let the planner generate solutions.

The Query Manager maintains five different kinds of information during planning: (1) an internal-state table $QM\_InternalState$, (2) a set of information sources $QM\_AvailableServices$, (3) a set of queries $QM\_PendingQueries$ for which one or more services have been invoked, but no answers have been received yet, (4) a set of queries $QM\_AnsweredQueries$ for which one or more services have returned the answers for them, and a set $QM\_QueriedNodes$ of pairs of the form (T, π) where T is a task network and π is a partial plan. Intuitively, the internal-state table $QM\_InternalState$ specifies a (possibly incomplete) state of the world for each node in the search tree of ENQUIRER. Each such state in this table is associated with the partial plan specified in the corresponding node of ENQUIRER's search tree. The set $QM\_QueriedNodes$ corresponds to the search nodes that ENQUIRER keeps track of in its $OPEN$ list. If a node (T, π) is in $QM\_QueriedNodes$ then this means that ENQUIRER requested information at least once while exploring that node during planning.

The set of information sources $QM\_AvailableServices$ is the set of Web Services given in the description of the current incomplete-information problem being solved. The Query Manager initializes $QM\_AvailableServices$ at the beginning of the planning process, and uses it every time a query needs to be directed to a Web Service. More specifically, when a query Q is issued by the planner, the query manager attempts to answer Q using the information in its internal state $QM\_InternalState(\pi)$ and the information available from $QM\_AvailableServices$.

In this paper, we only consider the information sources $I(\vec{x_i}, \vec{x_o}) \rightarrow D(\vec{y})$ where $\vec{x_i} \cup \vec{x_o} = \vec{y}$. This means that, for any substitution $\sigma_o \in R(I, \sigma_i)$, applying the input and output bindings $\sigma_o(\sigma_i(D))$ yields a set of ground atoms.

Using the routine QueryManager::IssueQuery, suppose ENQUIRER requests information about a conjunction C. Recall that C denotes the conjunction of (possibly unground) atoms that describes the preconditions of an HTN method or an operator. When invoked, the QueryManager::IssueQuery first checks whether the answers for any of the previously-issued queries have been received from the Web Services. More specifically, the procedure first checks if there is a previously-issued query of the form $Q(\vec{x}) \leftarrow C(\vec{x})$ such that the body of the query Q specifies the condition C. For each such query Q whose answer has been received, it removes Q from the set of previously-issued queries, inserts it in the set of answered queries, and updates the internal states of the Query Manager with the ground set of atoms generated by applying the variable substitutions to the specification of the service that returned the answer for Q.

An explanation about processing the incoming answers of the previously-issued queries is in order. As we described before, ENQUIRER issues queries for collecting information about the initial state of the world. Thus, when an answer is received for a query, we update each state in $QM\_InternalState$ such that the partial policy π' that is associated with that internal state does not contain any actions that deletes the information collected from the information source. This ensures the correctness of the Query Manager, and in turn, the ENQUIRER algorithm.

After processing the answers received for the pending queries, the QueryManager::IssueQuery routine performs the following checks on the atoms in C, the set of possibly unground atoms being queried. First, QueryManager::IssueQuery checks if C can be satisfied in the current incomplete internal state J of the planner — i.e., it checks the state generated by executing the actions in the current plan π in the initial state of the world. Note that QueryManager::IssueQuery performs this check only if it did not attempt to satisfy C in that current state before. If QueryManager::IssueQuery returned information about C from the state J before and ENQUIRER is still requesting information about C in this invocation of QueryManager::IssueQuery then this means that the information obtained from J did not help the planner to generate a solution. Therefore, more information from the relevant services must be gathered.

If the internal state for the current (T, π) pair does not provide information about C, then QueryManager::IssueQuery first checks whether there are pending queries for C – i.e., QueryManager::IssueQuery checks if there are any queries that has not been answered yet. If there are any, then the pending queries will provide a set of bindings for the variables in C so the Query Manager need not issue any queries about C. So, the Query Manager returns *wait*, forcing ENQUIRER to look for other plans along other search traces until the answers for those pending queries are received.

If there are no pending queries for C then the Query Manager checks whether there are already-answered queries for it. If there are any, since their answers have already been incorporated into every state in the $QM\_InternalState$, all the information about C is in $QM\_InternalState$. In this case, we first check if there is a substitution σ such that $\sigma(C)$ is ground and $\sigma(C)$ can be satisfied in the current state of the planner — i.e., if they can be satisfied in the state generated by executing the actions in the current plan π in the initial state of the world. If they cannot be satisfied in the current state of the world, then this means that one of more atoms in $\sigma(C)$ have been deleted by the actions π. In this case, C cannot be satisfied in the world, so the Query Manager return $\emptyset$ marking $failure$ (i.e., a backtracking point) for ENQUIRER.

If there are no pending and no already-answered queries about C, then this means that C consists of atoms that the Query Manager never encountered before. In this case, the Query Manager creates a query for C, finds and executes the services that are relevant to this query, and returns *wait* to ENQUIRER as before. In this paper, we do not address the problem of discovering the information sources relevant to a particular query. As we discuss in our related work, there are various techniques developed for this problem, and they can be incorporated in our framework easily. In our implementation of the ENQUIRER algorithm and the Query Manager, however, we assumed that if the body of a query can be satisfied in the body of an information source, then we matched the query to that information source. Note that this is correct since we require queries be always ground and information sources be always fully-specified, as described above.

7 Formal Properties of ENQUIRER

In this section, we discuss the formal properties of the ENQUIRER planning algorithm. The proofs of our theorems are given in the Appendix A.

We first establish the correctness of our algorithm. To do so, we will use the

```
procedure QueryManager::Initialize($J, X, T, \pi$)
 $QM\_AvailableServices \leftarrow X$
 $QM\_InternalState(\pi) \leftarrow J$
 $QM\_QueriedNodes \leftarrow \emptyset$
 $QM\_PendingQueries \leftarrow \emptyset$
 $QM\_AnsweredQueries \leftarrow \emptyset$
end-procedure

procedure QueryManager::IssueQuery($C, T, \pi$)
 for every query $Q(\vec{x}) \leftarrow C(\vec{x}) \in QM\_PendingQueries$ do
   if an $answer$ has been received for $Q$ from a service $I(\vec{x}) \rightarrow D(\vec{x})$ then
      remove $Q$ from $QM\_PendingQueries$
      insert $Q$ into $QM\_AnsweredQueries$
   for every variable substitution $\sigma \in answer$
      insert $\sigma(Y)$ into each entry $QM\_InternalState(\pi')$ such that $\pi'$ does not
         contain any action that deletes a ground atom in $\sigma(D)$
 unless $(T, \pi) \in QM\_QueriedNodes$ then
   $J \leftarrow QM\_InternalState(\pi)$
   $A \leftarrow \{\sigma \mid \sigma$ is a substitution such that $\sigma(C) \subseteq J\}$
   $QM\_QueriedNodes \leftarrow QM\_QueriedNodes \cup \{(T, \pi)\}$
   if $A \neq \emptyset$ then return $A$
 if there is a pending query in $QM\_PendingQueries$ for $C$ then return $wait$
 else if there are queries for $C$ in $QM\_AnsweredQueries$ then
   $J \leftarrow QM\_InternalState(\pi)$
   $A \leftarrow \{\sigma \mid \sigma$ is a substitution such that $\sigma(C) \subseteq J\}$
   return $A$
 else
   create a query $Q(\vec{x}) \leftarrow C(\vec{x})$
   find all services from $QM\_AvailableServices$ that can answer $Q$ and execute them
   return $wait$
end-procedure

procedure QueryManager::UpdateInternalState($\pi, action$)
 $S \leftarrow QM\_InternalState(\pi)$
 let $S'$ be the state that arises from applying $action$ in $S$
 $QM\_InternalState(\pi) \leftarrow S'$
end-procedure
```

Fig. 5. The Query Manager.

following lemma:

Lemma 4 *Let $P^I = (J, X, T, D)$ be an incomplete-information planning problem, and let $P^C = (S, T, D)$ be a complete-information planning problems that is consistent with P^I. Then, every search trace of the search tree for P^I is also a search trace in the search tree for P^C.*

The following theorem establishes the correctness of our approach.

Theorem 5 *Let $P^W = (J, W, C, K)$ be a service-composition problem, and $P^I = (J, X, T, D)$ be an incomplete-information planning problem that is equivalent to P^W. ENQUIRER returns a plan π for P^I if and only if π is a solution composition for P^W.*

Let $\chi(P^I)$ be the set of all solutions returned by any of the non-deterministic traces of ENQUIRER on an incomplete-information problem P^I. Furthermore, we let π_{P^I} be the shortest solution in $\chi(P^I)$, and let $|\pi_{P^I}|$ denote the length of that solution (i.e., plan). We now establish our first informedness theorem:

Theorem 6 *Let $P_1^I = (J_1, X_1, T, D)$ and $P_2^I = (J_2, X_2, T, D)$ be two incomplete-information planning problems. Then $\chi(P_1^I) \subseteq \chi(P_2^I)$, if $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$.*

A corollary immediately follows:

Corollary 7 *Let $P_1^I = (J_1, X_1, T, D)$ and $P_2^I = (J_2, X_2, T, D)$ be two incomplete-information planning problems. Then $|\pi_{P_2^I}| \leq |\pi_{P_1^I}|$, if $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$.*

Let $P^I = (J, X, T, D)$ be an incomplete-information planning problem. For the rest of this section, we will assume that there are constants c, p, q, m, and k such that c is the probability of a task being composite, p is the probability of a ground atom a being true, q is the probability of the truth-value of an atom a being known to ENQUIRER, m is the average number of methods that are applicable to a composite task, k is the average number of distinct preconditions for the methods in D, and d is the depth of the solution.

Without loss of generality, we assume that the task network T is a set of totally-ordered tasks. Furthermore, we assume that the search tree for a given incomplete-information problem is a complete tree whose depth is d, and there are no negated atoms in the preconditions of the methods in D.

Lemma 8 *Given an incomplete-information planning problem $P^I = (J, X, T, D)$ that satisfies the assumption given above, the probability ρ of ENQUIRER finding a solution for P^I is*

$$\rho_0 = 1; \text{ and}$$
$$\rho_d = (1 - c) * (p.q)^k + c * [1 - (1 - \gamma_d)^m],$$

where $\gamma_d = (p.q)^k \times \rho_{d-1}$.

The following theorem establishes our second informedness result.

Theorem 9 *Let $P_1^I = (J_1, X_1, T, D)$ and $P_2^I = (J_2, X_2, T, D)$ be two incomplete-information planning problems satisfying the assumption given ear-*

lier. Furthermore, let ρ_1 and ρ_2 be the probabilities of ENQUIRER *finding solutions for P_1^I and P_2^I, respectively. Then, $\rho_1 \leq \rho_2$, if $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$.*

8 Extending the Query Manager

The ENQUIRER planning algorithm and the Query Manager described so far are together designed to solve service-oriented planning problems under certain conditions of uncertainty. ENQUIRER is capable of gathering information during planning and explore different possible plans while the answers are generated by external information sources. However, ENQUIRER design is based on some restrictive assumptions that may create some difficulties in solving certain kinds of composition problems. In our study of such problems, we identified two additional key features that, we believe, are important for ENQUIRER to to handle. These key features are

- *Eager Planning.* When ENQUIRER requests information about a conjunction C, it issues a query to the Query Manager. The Query Manager finds and executes all of the services that can provide information about C, and returns that information only after all of those services are finished their executions. In many situations, however, it is possible for the planner to find a plan by using the information obtained from only some of those services. In such cases, the planner need not wait for all relevant services to finish their executions.
- *Conflicting Information.* In ENQUIRER, we assumed that the information gathered from the available Web Services is non-conflicting; e.g., when the Query Manager issues a query to two different Web Services, we assumed that the information returned by these services will always be consistent. In real world, however, this assumption does not always hold.

In the rest of this section, we focus on how to extend ENQUIRER to address these issues, and discuss the modifications to our original framework described above.

8.1 Extending the Query Manager for Eager Planning

Note that the Query Manager shown in Figure 5 returns all the information regarding a query Q at a search node (T, π) at once. That is, if X is the set of services that are relevant with a query Q, then the Query Manager executes all the services in X and returns the answers generated by these services as well as the answers that can be generated from the current internal state $QM\_InternalState(\pi)$. In other words, the answers returned by the Query

Manager regarding a query Q are generated as if $QM\_InternalState(\pi) \cup \delta(X)$ has been queried for Q all at once, where $\delta(X)$ denotes the total amount of information that can be obtained through querying the Web Services in X.

Although the Query Manager described above is provably sound and practical in many cases, it does not exploit a key feature of service-oriented planning: different Web Services that can answer a given query Q usually requires different execution times to return their answers, and the planner need not have all the possible information that can be gathered from those services to generate a plan. Therefore, it make sense for Query Manager to return the information gathered from the services invoked regarding Q as such information arrives from the services, and let the planner continue with planning as other services continue their execution. This way, if the planner fails to generate a plan with the answers from the finished services, then it backtracks and requests more information regarding Q, and as more information is received from the services, the Query Manager feeds the planner with that information as needed. Note that this requires the Query Manager to perform accessing, executing, and processing the information returned from the services in an iterative manner; contrary to the all-at-once approach as in its original description above.

In order to access and execute the Web Services that are relevant to a given query in such an iterative manner, we made the the following modifications to the ENQUIRER algorithm and the Query Manager of the Figures 3 and 5, respectively. The modification to the planning algorithm is as follows: The ENQUIRER algorithm keeps an $OPEN$ list as the set of nodes in the fringe of the search process. It performs its search by selecting a tuple from this list and by expanding it to generate the its successor in the search, and so on. Note that this way of using the $OPEN$ list ensures completeness in the planning process, when the planner receives all of the relevant information to its queries at once.

When the queries are answered in an iterative manner, it is not sufficient to keep only the fringe nodes of the search process in $OPEN$. Instead, we should keep every visited search node in $OPEN$, since the planner can backtrack any of those nodes and request for more answers for the queries it issued at those nodes. This requires the following simple modification in the ENQUIRER algorithm: After we issue a query Q and process the answer we received from the Query Manager regarding Q, we insert the current search node (T, π) into the $OPEN$ list again. This way, we keep all of the visited nodes in $OPEN$, enabling ENQUIRER to request more answers about the queries issued while visiting those nodes in an iterative manner. Note that, actually, we do not need to insert the search node (T, π) if the Query Manager tells ENQUIRER that there are no further answers about Q can be obtained from the available services — i.e., if the Query Manager returns $\emptyset$ when ENQUIRER invokes the QueryManager::IssueQuery procedure.

```
procedure QueryManager::IssueQuery($C, T, \pi$)
 for every query $Q(\vec{x}) \leftarrow X(\vec{x}) \in QM\_PendingQueries$ do
  if an $answer$ has been received for $Q$ from a service $I(\vec{x}) \rightarrow Y(\vec{x})$ then
    remove $Q$ from $QM\_PendingQueries$
    insert $Q$ into $QM\_AnsweredQueries$
  for every variable substitution $\sigma \in answer$
    insert $\sigma(Y)$ into each entry in $QM\_InternalState$

 unless $(T, \pi) \in QM\_QueriedNodes$ then
  $J \leftarrow QM\_InternalState(\pi)$
  $A \leftarrow \{\sigma \mid \sigma$ is a substitution such that $\sigma(C) \subseteq J\}$
  $QM\_QueriedNodes \leftarrow QM\_QueriedNodes \cup \{(T, \pi)\}$
  if $A \neq \emptyset$ then return $A$
 if there is a query for $C$ in $QM\_AnsweredQueries$ then
  $J \leftarrow QM\_InternalState(\pi)$
  $A \leftarrow \{\sigma \mid \sigma$ is a substitution such that $\sigma(C) \subseteq J\}$
  if $A = \emptyset$ and there are queries for $C$ in $QM\_PendingQueries$ then return $wait$
  return $A$
 else if there are queries for $C$ in $QM\_PendingQueries$ then return $wait$
 else
  create a query $Q(\vec{x}) \leftarrow C(\vec{x})$
  find all services from $QM\_AvailableServices$ that can answer $Q$ and execute them
  return $wait$
end-procedure
```

Fig. 6. The modified Query Manager for eager planning. The underlines indicate the modification made to the original algorithm of Figure 5.

Secondly, we modify the QueryManager::IssueQuery routine of the Query Manager as shown in Figure 6 where modifications are shown as underlined. Given a conjunction C to be queried, the modified QueryManager::IssueQuery procedure first checks if there are any queries that have been issued for C before and if the Query Manager has received answers for them. If this is the case, then we return those answers immediately, without waiting for all of the services executed for C complete their execution. If there are no already-answered queries for C but there are pending ones, then QueryManager::IssueQuery returns *wait*, forcing ENQUIRER to search for other solutions. If none of the above is the case, then QueryManager::IssueQuery creates a new query for C, finds and executes the services that can provide information about C, and returns the *wait* signal to ENQUIRER.

8.2 Dealing with Conflicting Information

In certain cases, the Web Services invoked to obtain an answer for a query may return conflicting answers. For example, in our doctor-patient example, a Web

Service can tell the planner that a particular time of the day is available for appointment in the hospital, whereas another service may specify that time slot as already occupied for another patient requiring the same treatment in the same hospital. In such cases, the planner must be able to resolve the conflicts in the information it gathers from the services, if possible, and generate a solution for the input planning problem under such conditions.

Our extensions to ENQUIRER to handle conflicting information during planning is as follows. First, we describe how to incorporate a way to detect conflicts in the Query Manager of Figure 5. To achieve this objective, we assume that the Query Manager is given a set of *integrity constraints*. These constraints can be defined as type and range constraints on certain arguments of a predicate, cardinality restrictions on some relationships, and so on. Note that, we do not have to restrict ourselves to only data integrity constraints. If a more expressive representation language such as OWL is being used, then the conflicts would be detected when the gathered data is not consistent with respect to the definitions in the ontologies.

When the Query Manager detects a conflict with the information returned by services for a given query, then it needs to attempt to resolve that conflict instead of bailing out immediately. In our framework, such a *conflict resolution* can be done as follows. We assume that a *ranking schema* is available for the Query Manager such that this schema specifies a *rank*, e.g. a positive real number, for each service in $QM\_AvailableServices$. This ranking schema can be implemented either as *static* or *context-dependent*. A static schema assigns a rank for each available service, and uses the same ranks for all queries issued to those services. A context-dependent schema, on the other hand, specifies a rank for the available services based on the query issued by the planner.

This simple sort of ranking scheme seems well suited for Web Service developers and easy to integrate with WSDL or OWL-S descriptions. Similar ranking schemes have been proposed for Web Service registries [7] to improve the Web Service discovery results. It is also possible to utilize the trust and reputation models designed for Web-based social networks [8]. For example, [9] proposes such a model to rank Web Services in a peer-to-peer framework. In a more complex system, the planner might store historical information and build a complex model for trust and reputation. The details of such systems are beyond the scope of this paper.

In our system, given a query Q, there is a a total-order that tells the Query Manager in which order to process the information from the services. The Query Manager, returns the answers received from the services to ENQUIRER starting with the answers of the highest-ranked service to that of the lowest-ranked service. Therefore, the Query Manager always inserts the set of ground atoms specified by the service into its internal state $QM\_InternalState$ based on this total-order, and can check whether the specified set of ground atoms are in conflict with this state. If a lower ranked service provides information conflicting with a higher ranked service the results are simply rejected. It is possible to accommodate more elaborate policies to resolve these conflicts, e.g. conflicts could be resolved differently when two high-ranked services are in disagreement compared to the case where a low-ranked service is in conflict with a high-ranked service. In either case, it would be easy to integrate such policies within the current framework and the algorithms of Figure 5 can be used in the presence of conflicting information without any further modifications.

9 Experimental Evaluation

For our experiments, we wanted to investigate (1) how well ENQUIRER would perform in service-composition problems where some of the information was completely unavailable, and (2) the trade-off between the time performance and the desirability of the solutions generated by the planner by using different query-processing strategies. We built a prototype implementation of the ENQUIRER algorithm that could be run with different query strategies. We also built a simulation program that would generate random response times for the queries issued by ENQUIRER. We ran our experiments on a Sun SPARC Ultra1 machine with 192MB memory running Solaris 8.

Experimental Case 1. In this case, we have investigated how the number of solutions (i.e., plans) found by the ENQUIRER algorithm is affected by the amount of the information available during planning. In these experiments, we used a set of Web Service composition problems on the Delivery domain described in Section 3. In these problems, a delivery company is trying to arrange the shipment of a number of packages by coordinating its several local branches. The company needs to gather information from the branches about the availability of vehicles (i.e., trucks and planes) and the status of packages. Such information is provided by Web Services, and the goal is to generate a sequence of confirmation messages that tells the company that every package is delivered to its destination.

In these experiments, we have ENQUIRER run the planner on 100 randomly-generated problem instances. For each problem instance, we ran ENQUIRER several times as described below, varying the amount of information available about the initial state by varying the following quantities: $|J|$, the number of atoms of S that were given initially; and $|\delta(X)|$, the amount of atoms of S that were made available through the input Web Services X, where S is the set of all possible ground atoms in the domain.

We measured the percentage of times that ENQUIRER could find plans, as a

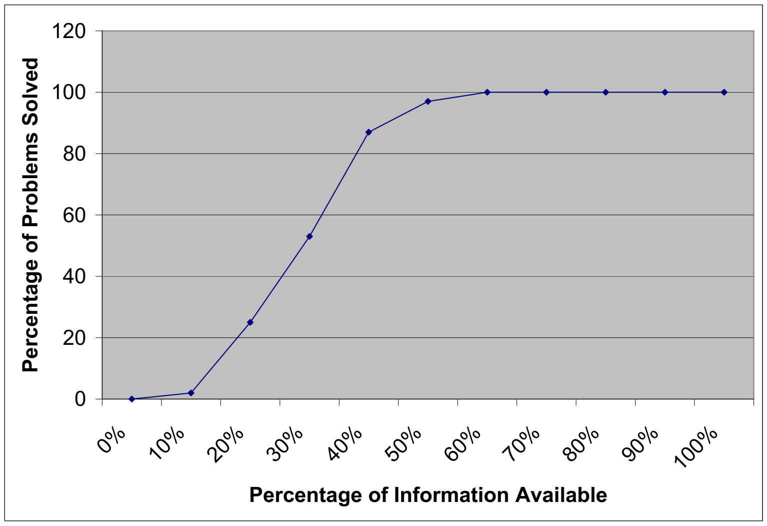

Fig. 7. The percentage of times ENQUIRER could find plans for the Delivery-Company problems, as a function of the amount of information available during planning.

function of the quantity $\frac{|J \cup \delta(X)|}{|S|}$. The fraction $\frac{|J \cup \delta(X)|}{|S|}$ is the fraction of atoms about the initial state that are available during planning. We varied $\frac{|J \cup \delta(X)|}{|S|}$ from $f = 0\%$ to 100% in steps of 10% as follows: we first randomly chose a set of atoms for $|S|$ such that the size of this set is equal to the fraction specified by the particular f value. Then, for each atom in this set, we randomly decided whether the atom should go in J – the incomplete initial state –, or it should be provided from the Web Services in X. Using this setup, we performed 100 runs of ENQUIRER for each value of $\frac{|J \cup \delta(X)|}{|S|}$. The results, shown in Fig.7, show that the success rate for ENQUIRER increased as we increased $\frac{|J \cup \delta(X)|}{|S|}$. ENQUIRER was able to solve 100% of the problem instances even when $\frac{|J \cup \delta(X)|}{|S|}$ was as low as 60%.

Experimental Case 2. We also aimed to investigate the comparison between the time performance of the non-blocking nature of the search performed by the ENQUIRER algorithm and the desirability of the plans generated — i.e., how much the generated plans conforms to the user preferences encoded in the domain descriptions given to the algorithm in terms of the ordering among the methods applicable to the same task.

In this respect, we have implemented three variations of the ENQUIRER algorithm. All of these variations are based on how the planning algorithm maintains its $OPEN$ list. The original algorithm, shown in Figure 3, always searches for solutions that can be found regardless of the incompleteness of the information about the initial state: when ENQUIRER issues a query, it does not wait for an answer; instead it continues its search for solutions that can be found without issuing any queries at all. We call this strategy as the

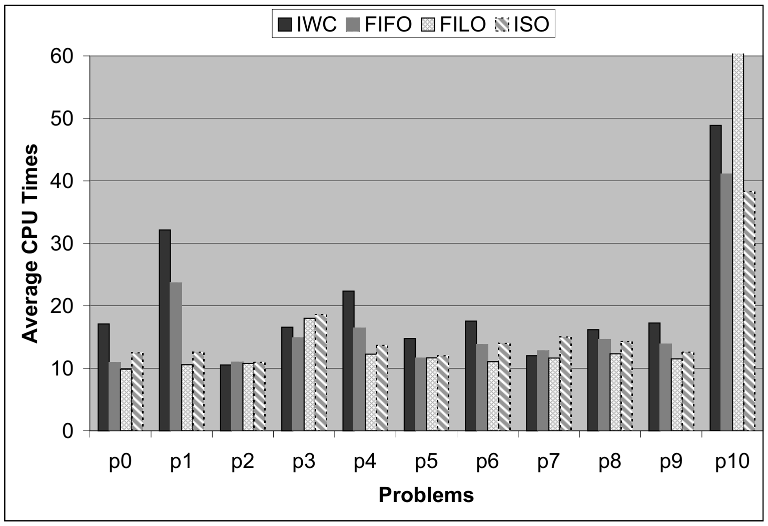

Fig. 8. The comparison of the average CPU times (in secs.) of the four search strategies in ENQUIRER.

Issue-Search-Other (ISO) strategy. Similar to ISO, all the other three strategies are also based on different implementations of the OPEN list. The Issue-Wait-Continue (IWC) strategy involves blocking the search until a query, which is issued recently, gets answered. In the First-In-Last-Out (FILO) strategy, if the planner has asked many queries during its search and several of them have been answered, then it always returns to the point in the search space that corresponds to the most recently-issued query that has been answered. Finally, in FIFO which is a complete symmetric version of FILO, the planner always considers the answers corresponding to the least-recent query it has asked.

In our experiments with these variations of ENQUIRER, we compared their time performances on the domain used in [2], which is partly based on the scenario described in the Scientific American article about the Semantic Web [3]. This scenario describes two people who are trying to take their mother to a physician for a series of treatments and follow-up meetings. The planning problem is to come up with a sequence of appointments that will fit in to everyone's schedules, and at the same time, to satisfy everybody's preferences.

We have created 11 random problems in this domain and we have run all strategies 10 times in each problem. We have set the maximum response time for a query asked by ENQUIRER to be 1 seconds. Fig.8 reports the average CPU times required by the four strategies described above.

These results provide several insights regarding the trade-off between time performance of ENQUIRER and the desirability of the solutions it generates under incomplete information. As we described above, in our framework, the desirability of a solution is determined by the set of methods available for the nonprimitive tasks that are provided to ENQUIRER and the order specified over the methods for a particular task provided to the planner.

From a performance-oriented point of view, our results showed that it is very important for the planner not to wait until its queries get responded, especially when it takes a very long time to get responses for those queries. In our experiments, the IWC strategy performed worst than the others in most of the problems. However, IWC was able to generate the most-desirable solutions for our problems, since it always considers the methods in ENQUIRER knowledge base in the order they were specified.

In most of the problems, the FILO strategy was able to find solutions more quickly than the other strategies; in particular, it was consistently more efficient than IWC. However, these solutions were usually less-desirable ones with respect to the user preferences encoded as orderings among the methods in ENQUIRER knowledge base, since FILO always tends to consider the answers for the most-recently issued queries, which usually correspond to less-desirable preferences because of the queries waiting for an answer.

Our experiments also suggest that the ISO strategy can be particularly useful on planning problems that have few solutions. In particular, the problem $p10$ of our experiments is one such problem, and ISO was able to find solutions for this problem more quickly than others. The reason for this behavior is that ISO performs a less organized and less structured search compared to other strategies, enabling the planner to consider different places in the search space. On the problem $p10$ which has only 48 solutions compared to the 4248 solutions of the problem $p9$, this characteristic enabled ISO to perform better than the other strategies. However, in most of the problems, the solutions generated by the planner using ISO were not the desirable ones.

10 Related Work

The first approach for Web Service Composition, described in [5], is based on the notion of generic procedures and customizing user constraints. This work extends the Golog language to enable programs that are generic, customizable and usable in the context of the Web. They also successfully augment a ConGolog interpreter that combines online execution of sensing actions with offline simulation of world altering services. Our approach is very similar to this in spirit, with the following two key differences 1) We allow the planner to change (i.e. simulate the changes) the information gathered from external sources whereas the Invocation and Reasonable Persistence (IRP) assumption of [5] prevents such changes. Note that, however, we still require that no agent other than the planner can change the state of the world during planning. 2) We use more flexible information source descriptions where the information returned from the service is described as a conjunction of atoms. On the other hand, [5] relates single fluents to one or more external service calls. Having more than one atom in the body of source descriptions enables us to describe a wider range of information sources and answer more complex queries. Also, note that, the approach described in [5], is based on situation calculus and such a general logic-based system has the ability to do arbitrary reasoning about first-order theories. This is important because planning with OWL-S service descriptions requires a reasoner capable of handling the expressivity of OWL. However, Golog implementation uses regression to reason about actions, i.e. to solve executability and projection problems. As discussed in detail in [10], translating OWL-S descriptions (or descriptions of similar expressitivity) to situation calculus and applying regression yields a standard first-order theory which is not in the scope of what Golog can handle without calling a general first-order theorem prover. On the other hand, integrating an OWL reasoner with an HTN planner is a relatively easy task (since the planner and the reasoner are modular components) and such integration has been shown to be practical and efficient [11].

The first work that enables us to use HTN-planning techniques for solving service-composition problems is described in [6]. This work presents a formal model of the relationship between the OWL-S process ontology [1] used for describing Web Services and *Hierarchical Task Networks* as in SHOP2 planning system [4]. In this formulation, the *atomic* and *composite* OWL-S processes are mapped into *primitive* and *nonprimitive* tasks in SHOP2, and the planner is used for generating a collection of atomic process instances that, when executed in the initial state of the world, achieves the desired functionality. In this paper, we extend this approach to be able to cope better with the fact that information-providing Web Service may not return the needed information immediately when they are executed, or at all. In particular, our ENQUIRER algorithm does not cease the search process while waiting answers to some of its queries, but keeps searching for alternative compositions that does not depend on answering those specific queries.

Another planning approach is a technique proposed by [12]. This technique is based on an *estimated-regression* planner, called Optop, that is used for generating compositions for WSC problems. In Optop, a state of the planner is a *situation*, which is essentially the current partial plan. Optop works with classical-planning goals; thus, it checks whether the current situation satisfies the conjunction of the goal literals given to the planner as input. During its search, Optop computes a *regression-match graph* as described in [12], which essentialy provides information about how to reach to a goal state from the current situation. The planner returns the successor situations that arises from applying the actions specified by that graph in the current situation. Note that this technique is very different than our approach since, in Optop, only the precondition and effect information about services are used to generate compositions whereas our HTN-based approach generates plans using the structure of composite services.

In a different approach, [13] proposed to model the services and the information about the world by using the "knowledge-level formulation" first introduced in the PKS planning system [14]. This formulation involves modeling Web Services based on not what is actually true or false about them, but what the agent that performs the composition actually knows to be true or false about their operations and the results of those operations. In this approach, a composition is formulated as a conditional plan, which allows for interleaving the executions of information-providing and world-altering services, unlike the works described above.

Another approach for Web Service composition is proposed in [15] and [16]. This approach is a planning technique based on the "Planning as Model Checking" paradigm for the automated composition of web services described in OWL-S process models. The OWL-S process models are translated into state transition systems that describe the dynamic interactions with external services. The composition goals are expressed in a language where temporal restrictions on goals and preferences about goals can also be specified. With the composition goal and the state transition systems, the planner, based on symbolic model checking techniques, returns an executable process rather than a linear sequence of actions.

An important difference between the works described in [13,15,16] and our approach is that we do not model the information sources as sensing actions and generate plans that include branching points which correspond to the possible outcomes of those sensing action that will be observed when the plan is actually executed. Instead, following [5,6], we assume that the information sources are executed during planning. As a result, ENQUIRER can generate simple plans since observational actions are not included in the plan, and it can interact with external information sources and clear out the "unknown"s during planning time as much as possible. As a future work, we are planning to investigate a hybrid approach where some information is gathered during planning and the remaining information is gathered by sensing actions that are placed in the generated conditional plan. Such an approach would be more robust to changes in the environment because the modifications done by other agents would not necessarily invalidate the plans generated.

In the AI planning literature, there are various approaches to planning with incomplete-information, designed for gathering information during execution by inserting sensing actions in the plan during planning time and by generating conditional plans conditioned on the possible pieces of information that can be gathered by those actions. In addition to the ones mentioned above, examples include the following. [17] presents a planning language called UWL, which is an extension of the STRIPS language, in order to distinguish between (1) the world-altering and the observational effects of the actions, and (2) the goals of satisfaction and the goals of information. [18] describes he XII planner for planning with both complete and incomplete information, which is an extension of UCPOP [19], and is able to generate sensing actions for information gathering during execution.

There are various algorithms developed for information integration problems [20–22] which have recently been applied to the problem of composing information-providing services [23]. These algorithms find and combine a set of information sources to answer a given query. Note that, this is the same task ENQUIRER's Query Manager needs to perform to answer the issued queries. In our work, we have not specifically addressed this problem because these existing algorithms can easily be integrated in our system. We note that this integration is straight-forward as our definitions of a query and an information source aligns with the existing work.

A speculative execution method is described in [24] for generating information-gathering plans in order to retrieve, combine, and manipulate data located in remote sources. This technique exploits certain hints received during plan execution to generate speculative information for reasoning about the dependencies between operators and queries later in the execution. ENQUIRER differs from this method in two aspects: (1) it searches different branches of the search space when one branch is blocked with a query execution, and (2) it runs the queries in planning time, whereas speculative execution was shown to be useful for executing plans. Combining speculative execution with our approach would enable us to run queries down in a blocked branch; however, since the time spent for a query is lost when speculations about that query are not valid, it is an open question if combining the two approaches will lead to significant results.

11 Conclusions and Future Work

In our previous work [2], we have shown how a set of OWL-S service descriptions can be translated to a planning domain description that can be used by SHOP2. The corresponding planning problem for a service composition problem is to find a plan for the task that is the translation of a composite process. This approach differentiates between the information-gathering services, i.e. services that have only outputs but no effects, and the world-altering services, i.e. services that have effects but no outputs. The preconditions of information-gathering services include an external function to call to execute the service during planning and add the information to the current state. This can be seen as a specialized application of the ENQUIRER algorithm, when the queries are explicitly specified as special primitive tasks that correspond to atomic service executions.

ENQUIRER overcomes the following limitations in our previous work:

- The information providing services do not need to be explicitly specified in the initial description. The Query Manager can be used to select the appropriate Web Service on the fly when the information is needed. Note that matching service description does not need to be an atomic service. A composite service description matching the request could be recursively fed to the planner, or an entirely different planner could be used to plan for information gathering.
- The planning process does not need to wait for the information gathering to finish and can continue planning while the service is still executing. Furthermore, the Query Manager can collect and return the answers for those queries as they are received from the information sources to ENQUIRER, enabling the planner to perform an eager search without waiting all of the information regarding its queries and asking more information only if it needs it to generate solutions.
- The information sources may provide conflicting answers to ENQUIRER's queries. In our framework, it is the Query Manager's responsibility to detect and resolve such conflicts. In this work, we assumed that the Query Manager is equipped with a mechanism that ranks the available Web Services, and it processes the information received from those services based on their ranking. As we described in the paper, this provides a simple, yet effective, technique for conflict detection and resolution.

In this paper, we have assumed that information-providing services cannot have world-altering effects. Otherwise, the correctness of the plans generated cannot be guaranteed since the changes done by the information-providing service may invalidate some of the steps planner already committed to. However, this restriction is not necessary when the effects of the information-gathering services do not interact with the plan being sought for. As an example, consider a service that charges a small amount of fee for the service. If we are looking for a plan that has nothing to do with money, then it would be safe to execute this service and change the state of the world. In general, this safety can be established when the original planning problem and information-gathering problem correspond to two disconnected task networks that can be accomplished without any kind of interaction. Verifying that there is no interaction between two problems is a challenging task that we will address in our future work.

ENQUIRER is designed for information gathering at plan time; however, it is important to do so in execution time as well. We hypothesize that it should possible to extend our framework for this purpose as follows. ENQUIRER's queries are about the initial state of a planning problem, to ensure that the planner is sound and complete. However, in principle, we should be able to issue queries about any state during planning. This would allow us to insert queries, similar to sensing actions, in the plan generated by the planner, leading conditional plans to be generated by the planner based on the possible answers to such queries. We are currently exploring this possibility and its ramifications on planning.

Acknowledgments

We would like to thank our anonymous reviewers for their insightful comments that were very helpful in preparing the final version of this paper.

This work was supported in part by the following grants, contracts, and awards: NSF grant IIS0412812, Air Force Research Laboratory F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, the Defense Advanced Research Projects Agency (DARPA), Fujitsu Laboratory of America at College Park, Lockheed Martin Advanced Technology Laboratories, the National Science Foundation (NSF), the National Institute of Standards and Technology (NIST), and NTT Corp. The opinions expressed in this paper are those of authors and do not necessarily reflect the opinions of the funders.

References

[1] OWL Services Coalition, OWL-S: Semantic markup for web services, OWL-S White Paper http://www.daml.org/services/owl-s/0.9/owl-s.pdf (2003).

[2] D. Wu, B. Parsia, E. Sirin, J. Hendler, D. Nau, Automating daml-s web services composition using SHOP2, in: Proceedings of 2nd International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, 2003.

[3] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American 284 (5) (2001) 34–43.

[4] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, F. Yaman, SHOP2: An HTN planning system, Journal of Artificial Intelligence Research 20.

[5] S. McIlraith, T. Son, Adapting Golog for composition of semantic web services, in: Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning, Toulouse, France, 2002.

[6] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau, HTN planning for web service composition using SHOP2, Journal of Web Semantics 1 (4) (2004) 377–396. URL http://www.mindswap.org/papers/SHOP-JWS.pdf

[7] I. Constantinescu, W. Binder, B. Faltings, An extensible directory enabling efficient semantic web service integration, in: International Semantic Web Conference, 2004, pp. 605–619.

[8] J. Golbeck, B. Parsia, J. Hendler, Trust networks on the semantic web, in: Proceedings of Cooperative Intelligent Agents 2003, Helsinki, Finland, 2003.
URL http://www.mindswap.org/papers/CIA03.pdf

[9] F. Emekci, O. D. Sahin, D. Agrawal, A. E. Abbadi, A peer-to-peer framework for web service discovery with ranking, in: IEEE International Conference on Web Services (ICWS'04), 2004, pp. 192–199.

[10] F. Baader, C. Lutz, M. Milicic, U. Sattler, F. Wolter, Systematic nonlinear planning, in: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, 2005.

[11] E. Sirin, B. Parsia, Planning for semantic web services, in: Semantic Web Services Workshop at 3rd International Semantic Web Conference (ISWC2004), Hiroshima, Japan, 2004.
URL http://www.mindswap.org/papers/SWS-ISWC04.pdf

[12] D. McDermott, Estimated-regression planning for interactions with web services, 2002, pp. 204–211.

[13] E. Martinez, Y. Lespérance, Web service composition as a planning task: Experiments using knowledge-based planning, in: Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services, 2004, pp. 62–69.

[14] R. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: In Proceedings of the Fourth International Conference on AI Planning and Scheduling (AIPS'98), 1998, pp. 86–93.

[15] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, P. Traverso, Planning and monitoring web service composition, in: The 11th International Conference on Artificial Intelligence, Methodologies, Systems, and Applications (AIMSA), 2004, pp. 106–115.

[16] P. Traverso, M. Pistore, Automated composition of semantic web services into executable processes., in: International Semantic Web Conference, 2004, pp. 380–394.

[17] O. Etzioni, D. Weld, D. Draper, N. Lesh, M. Williamson, An approach to planning with incomplete information, in: Proceedings of KR-92, 1992.

[18] K. Golden, O. Etzioni, D. Weld., Planning with execution and incomplete information, Tech. Rep. TR96-01-09, Department of Computer Science, University of Washington (February 1996).

[19] J. S. Penberthy, D. Weld, UCPOP: A Sound, Complete, Partial Order Planner for ADL, in: Proceedings of KR-92, 1992.

[20] O. M. Duschka, A. Y. Levy, Recursive plans for information gathering., in: IJCAI (1), 1997, pp. 778–784.

[21] A. Y. Levy, A. Rajaraman, J. J. Ordille, Querying heterogeneous information sources using source descriptions, in: Proceedings of the Twenty-second International Conference on Very Large Databases, VLDB Endowment, Saratoga, Calif., Bombay, India, 1996, pp. 251–262.

[22] A. Y. Levy, A. Rajaraman, J. J. Ordille, Query-answering algorithms for information agents., in: AAAI/IAAI, Vol. 1, 1996, pp. 40–47.

[23] J. L. A. Snehal Thakkar, C. A. Knoblock, A data integration approach to automatically composing and optimizing web services, in: In Proceeding of 2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[24] G. Barish, C. A. Knoblock, Planning, executing, sensing, and replanning for information gathering, in: Proceedings of Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002), AAAI Press, Menlo Park, CA, 2002, pp. 184–193.

A Proofs of Our Theorems

Lemma 4 *Let $P^I = (J, X, T, D)$ be an incomplete-information planning problem, and let $P^C = (S, T, D)$ be a complete-information planning problems that is consistent with P^I. Then, every search trace of the search tree for P^I is also a search trace in the search tree for P^C.*

PROOF. Let $P^I = (J, X, T, D)$ be an incomplete-information planning problem, and let $P^C = (S, T, D)$ be a complete-information planning problem that is consistent with P^I. By the definition of consistency, we have $J \cup \delta(X) \subseteq S$. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the search trees for the problems P^I and P^C, respectively.

The proof is by induction on the length d of a search trace in a search tree. Let $d = 0$. Obviously, this marks the search node that is the root of the trees $\mathcal{T}_1$ and $\mathcal{T}_2$. Note that both $\mathcal{T}_1$ and $\mathcal{T}_2$ have the same root node since both problems P^I and P^C specify the same goal task network T. This establishes the base case for our induction.

Now, assume that every search trace with length d in $\mathcal{T}_1$ is also in $\mathcal{T}_2$. Let $n = (T, \pi)$ be a leaf node of a search trace with length d. If $T = nil$ then n is a leaf node in the search tree for both P^I and P^C. Suppose $T \neq nil$ and let t be a task that has no predecessors in the task network in T. If t is primitive and there is an operator o for it in D, then both $\mathcal{T}_1$ and $\mathcal{T}_2$ will have the same search node as a child of n at depth $d + 1$, where this child of n will mark the task network obtained by removing t from the task network of n. If t is nonprimitive, the set of methods applicable to t given $J \cup \delta(X)$ is a subset of the set of methods applicable to t given S, since $J \cup \delta(X) \subseteq S$. Then, it follows that the set of child nodes of n in $\mathcal{T}_1$ is a subset of the child nodes of n in $\mathcal{T}_2$ at depth $d+1$ in both trees, and therefore, the set of search traces going out from n in $\mathcal{T}_1$ is a subset of those going out from n in $\mathcal{T}_2$. This completes the proof of the theorem.

Theorem 5 *Let $P^W = (J, W, C, K)$ be a service-composition problem, and $P^I = (J, X, T, D)$ be an incomplete-information planning problem that is equivalent to P^W. ENQUIRER returns a plan π for P^I if and only if π is a solution composition for P^W.*

PROOF. Suppose $P^W = (J, W, C, K)$ is a service-composition problem and suppose $P^I = (J, X, T, D)$ is an incomplete-information planning problem that is equivalent to P^W. Then, by the definition of equivalency, we have $\delta(X) = \delta(W)$, T is the SHOP2 translation for the OWL-S process C, and D = TRANSLATE-PROCESS-MODEL(K) is the HTN-domain description generated by the translation algorithm of [6].

The proof proceeds by showing that there is a bijection between the set of solution plans for P^I and the set of solution compositions for P^W. First of all, we define the *complete* version of a service-composition problem $P^W = (J, W, C, K)$ as the problem $P^{WC} = (S, C, K)$ such that $S = J \cup \delta(W)$. That is, in the complete version of P^W, we have in the initial state S all the information that can be gathered through the Web Services in W. Clearly, the set of solutions for P^W is the same as the set of solutions for P^{WC} since both problems uses the same total amount of information about the world, the same goal process C, and the same set of process models K.

Let $P^C = (S, T, D)$ be the complete-information planning problem corresponding to $P^{WC} = (S, C, K)$. By the Theorem 5 of [6], there is a bijection between the solutions for P^C and P^{WC}. Then,it follows that the set of solutions for the complete-information planning problem $P^C = (S, T, D)$ and the incomplete-information planning problem $P^I = (J, X, T, D)$ are the same, since we have $S = J \cup \delta(X)$ by Lemma 4. Therefore, it follows that there is a bijection between the set of solutions for P^I and the set of solutions for P^W.

Theorem 6 *Let $P^I_1 = (J_1, X_1, T, D)$ and $P^I_2 = (J_2, X_2, T, D)$ be two incomplete-information planning problems. Then $\chi(P^I_1) \subseteq \chi(P^I_2)$, if $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$.*

PROOF. Let $\chi(P^I)$ be the set of all solutions returned by any of the nondeterministic traces of ENQUIRER on an incomplete-information problem P^I. Furthermore, we let π_{P^I} be the shortest solution in $\chi(P^I)$, and let $|\pi_{P^I}|$ denote the length of that solution (i.e., plan).

Let $P^I_1 = (J_1, X_1, T, D)$ and $P^I_2 = (J_2, X_2, T, D)$ be two incomplete-information planning problems. Suppose we have $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$. Furthermore, let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the search trees for the problems P^I_1 and P^I_2, respectively.

The proof starts by showing that the set of search traces of the search tree $\mathcal{T}_1$ is a subset of those of the tree $\mathcal{T}_2$. Therefore, the set of successful search traces in $\mathcal{T}_1$ — i.e., the set of search traces that ends in a node that specifying the empty task network, and therefore, that marks a solution plan — is also a subset of the set of successful search traces in $\mathcal{T}_2$, which means that we have $\chi(P^I_1) \subseteq \chi(P^I_2)$.

The proof is by induction on the depth d of a search trace. Let $d = 0$. Obviously, this marks the search node that is the root of the trees $\mathcal{T}_1$ and $\mathcal{T}_2$. Note that both $\mathcal{T}_1$ and $\mathcal{T}_2$ have the same root node since both problems P^I_1 and P^I_2 specify the same goal task network T. This establishes the base case for our induction.

Now, assume that every search trace with depth d in $\mathcal{T}_1$ is also in $\mathcal{T}_2$. Let $n = (T, \pi)$ be a leaf node of a search trace with length d. If $T = nil$ then n is a terminal node in the search tree for both P^I and P^C. Suppose $T \neq nil$ and let t be a task that has no predecessors in the task network in n. If t is pritimive and there is an operator o for it in D, then both $\mathcal{T}_1$ and $\mathcal{T}_2$ will have the same search node as a child of n at depth $d+1$, where this child of n will mark the task network obtained by removing t from the task network of n. If t is nonprimitive, the set of methods applicable to t is a subset of the set of methods applicable to t, since $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$. Then, it follows that the set of child nodes of n in $\mathcal{T}_1$ is a subset of the child nodes of n in $\mathcal{T}_2$ at depth $d+1$ in both trees. Therefore, it follows that every search trace of $\mathcal{T}_1$ is also a search trace in $\mathcal{T}_2$.

Corollary 7 *Let $P_1^I = (J_1, X_1, T, D)$ and $P_2^I = (J_2, X_2, T, D)$ be two incomplete-information planning problems. Then $|\pi_{P_2^I}| \leq |\pi_{P_1^I}|$, if $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$.*

PROOF. Immediate from Theorem 6.

Lemma 8 *Given an incomplete-information planning problem $P^I = (J, X, T, D)$ that satisfies the assumption given above, the probability ρ of ENQUIRER finding a solution for P^I is*

$$\rho_0 = 1; \quad and \tag{A.1}$$
$$\rho_d = (1-c) * (p.q)^k + c * [1 - (1-\gamma_d)^m], \tag{A.2}$$

where $\gamma_d = (p.q)^k \times \rho_{d-1}$.

PROOF. Let $P^I = (J, X, T, D)$ be an incomplete-information planning problem. Suppose that there are constants c, p, q, m, and k such that c is the probability of a task being composite, p is the probability of a ground atom a being true, q is the probability of the truth-value of an atom a being known to ENQUIRER, m is the average number of methods that are applicable to a composite task, and k is the average number of distinct preconditions for the methods in D.

Without loss of generality, suppose that the task network T is a set of totally-ordered tasks, and that the search tree for a given incomplete-information problem is a complete tree whose depth is d, and there are no negated atoms in the preconditions of the methods and operators in D.

Note that ENQUIRER finds a solution for P^I if and only if there is a search trace that starts from the root node of the search tree for P^I and ends at a leaf node that marks the empty task network. Then, it follows that the probability that ENQUIRER finds a solution for P^I is equal to the probability that a search trace in the search tree for P^I reaches to a leaf node, starting from the root node of the tree. Below, we show that this probability is given by the recurrence relation of the Lemma 8.

The proof is by induction on the depth of a solution. We define d as the *depth-to-go* value for a search node (T, π) in the search trace of ENQUIRER that corresponds to a solution. That is, d is the number of node that the planner visits after expanding (T, π) until it reaches to the leaf node with the empty task network. If $d = 0$ then we have $T = nil$ for the input planning problem. In this case, ENQUIRER will return the empty plan immediately. Therefore, the probability of finding the empty plan as the solution for the input planning problem is 1 as given in Eqn A.1. This establishes the base of our inductive proof.

Now assume that the theorem correctly gives the probability of reaching to a leaf node, starting from a node with a depth-to-go value d. Now consider a search trace with depth $d+1$ whose leaf node specifies the empty task network. Let T be the goal task network that this search trace accomplishes and let $J' \cup \delta(X)$ be the amount of information available to accomplish T. In other words, T is the task network in the start node of this trace and J' is the incomplete state for that search node. Note that J' is defined by applying the actions in π to the incomplete initial state J given the information gathered along this search trace from the services in X. Furthermore, let t be the first task T — recall that T is a totally-ordered task network by our assumptions stated earlier. By the same assumptions, we know that t is a primitive task with the probability $(1 - c)$ and it is nonprimitive with probability c. In the former case, the probability that t will be accomplished is $(p.q)^k$;i.e., the probability that the preconditions of the operator that matches to the task t is satisfied in $J \cup \delta(X)$.

The case in which t is not primitive is more complicated. Suppose that there are m methods that matches to t. Let γ_{d+1} be the probability that each such method will be successfully applied to t. Note that a method will be successfully applied iff (1) its preconditions will be satisfied in $J' \cup \delta(X)$, and (2) the task network that will be generated by removing t from T and adding the subtasks for t. Note that this task network is a search node in the search tree for the input problem with a depth d. By the induction hypothesis, the probability of that node is given by the theorem. Therefore, the probability that a method will be successfully applied to t with a depth $d+1$ is $\gamma_{d+1} = (p.q)^k.\rho_d$.

Then, assuming that the application of each method is independent from each other, the probability that t will be decomposed successfully into subtasks is $c.[1 - (1 - \gamma_{d+1})^m]$. Therefore, the probability of t will be accomplished is

$(1-c).(pq)^k + c.[1-(1-\gamma_{d+1})^m]$. This completes the proof of the theorem.

Theorem 9 *Let* $P_1^I = (J_1, X_1, T, D)$ *and* $P_2^I = (J_2, X_2, T, D)$ *be two incomplete-information planning problems satisfying the assumption given earlier. Furthermore, let* ρ_1 *and* ρ_2 *be the probabilities of* ***ENQUIRER*** *finding solutions for* P_1^I *and* P_2^I*, respectively. Then,* $\rho_1 \leq \rho_2$*, if* $J_1 \cup \delta(X_1) \subseteq J_2 \cup \delta(X_2)$*.*

PROOF. Immediate from Corollary 7 and Lemma 8.