Web Service Composition with Volatile Information

Tsz-Chiu Au, Ugur Kuter, and Dana Nau

Department of Computer Science and Institute for Systems Research,
University of Maryland, College Park, MD 20742, USA
{chiu, ukuter, nau}@cs.umd.edu

Abstract. In many Web service composition problems, information may be needed from Web services during the composition process. Existing research on Web service composition (WSC) procedures has generally assumed that this information will not change. We describe two ways to take such WSC procedures and systematically modify them to deal with volatile information.

The *black-box* approach requires no knowledge of the WSC procedure's internals: it places a wrapper around the WSC procedure to deal with volatile information. The *gray-box* approach requires partial information of those internals, in order to insert coding to perform certain bookkeeping operations.

We show theoretically that both approaches work correctly. We present experimental results showing that the WSC procedures produced by the gray-box approach can run much faster than the ones produced by the black-box approach.

1 Introduction

Most existing research on automated composition of semantic Web services has focused on *Web service composition* (WSC) procedures, i.e., procedures for finding a composition of Web services to accomplish a given task. In order to assemble a composition, a WSC procedure itself may need to retrieve information from Web services while it is operating. Existing works have generally assumed that such information is *static*, i.e., it will never change. For example, the Golog-based [1] and HTN-based [2, 3] approaches both use the Invocation and Reasonable Persistence (IRP) condition. The WSC procedures reported in [4, 5] are even more restrictive: they require that all of the information needed by their procedures is provided by the user as input parameters. We will refer to such procedures as *static-information* WSC procedures.

Clearly there are many cases where the static-information assumption is unrealistic. There are thousands of Web services whose information may change while a WSC procedure is operating: for example, whether a product is in stock, how much it will cost or how much has been bid for it, what the weather is like, what time a train or airplane will arrive, what seats are available for an airplane or a concert, what resources are available in a grid-computing environment, and so forth.

This paper focuses on how to take static-information WSC procedures such as the ones mentioned above, and translate them into *volatile-information* WSC procedures that work correctly when information obtained from Web services may change.

Our primary contributions are as follows:

1. We provide a general procedural model for a class of WSC procedures. We model them as trial-and-error search procedures that may try different possible Web service compositions in order to find one that accomplishes the desired task.
2. We describe a *black-box approach* for translating static-information WSC procedures into volatile-information WSC procedures. In particular, we describe a wrapper that can be placed around any WSC procedure, without needing to know how the underlying composition procedure operates.
3. We describe a *gray-box approach* for translating static-information WSC procedures into volatile-information WSC procedures. This approach is based on taking our procedural model mentioned above, and modifying it to deal with volatile information—hence the same modification will work on any WSC procedure that is an instance of our general procedural model. We call this approach a *gray-box* approach because it requires *partial* knowledge about a WSC procedure: namely, that the WSC procedure is an instance of our procedural model.
4. We state theorems saying that both the black-box and the gray-box approaches work correctly on any WSC procedure that is an instance our general model.
5. We provide experimental results demonstrating that the gray-box approach produces volatile-information WSC procedures that may run exponentially faster than the ones produced by the black-box approach. For example, in a set of problems in which there were only seven information items that needed to be retrieved from Web services, the procedure produced by the gray-box approach ran 50 times as fast as the one produced by the black-box approach.

It also would be possible to define a *white-box* approach, namely to take the code for the WSC procedure and rewrite it by hand. But this approach would be labor-intensive and it would only extend a single composition procedure, hence we do not consider it in this paper. Our results show that in comparison with the black-box approach, the gray-box approach already can provide substantial speedups without having to delve into all of the details of the original WSC procedure.

2 Procedural Model of Web Service Composition

Existing approaches for Web Service Composition formulate the problem in different ways, depending mainly on how the developers of those approaches perceive the problem. Examples include the following:

- In [1], the states of the world and the world-altering actions are modeled as Golog programs, and the information-providing services are modeled as external functions calls made within those programs. The goal is stated as a Prolog-like query and the answer to that query is a sequence of world-altering actions that achieves the goal, when executed in the initial state of the world. During the composition process, however, it is assumed that no world-altering services are executed. Instead, their effects are simulated in order to keep track of the state transitions that will occur when they are actually executed.

- In [2], the WSC procedure is based on the relationship between the OWL-S process ontology [6] used for describing Web services and *Hierarchical Task Networks* as in HTN Planning [7]. OWL-S processes are translated into tasks to be achieved by the SHOP2 planner [7], and SHOP2 generates a collection of atomic process instances that achieves the desired functionality.
- [3] extends the work in [2] to cope better with the fact that information-providing Web services may not return the needed information immediately when they are executed, or at all. The ENQUIRER algorithm presented in this work does not cease the search process while waiting answers to some of its queries, but keeps searching for alternative compositions that do not depend on answering those specific queries.
- [4] models Web services and information about the world using the "knowledge-level formulation" first introduced in the PKS planning system [8]. This formulation models Web services based not on 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. 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.

Despite their differences, the aforementioned approaches have the following features in common:

1. The WSC procedure is given the specification of the Web services written in a formal language such as OWL-S [6], and a goal to be accomplished in the world.
2. The WSC procedure does a trial-and-error search through some space of possible solutions, to try to find a complete solution. A *solution* for a Web service-composition problem is a set of services with ordering constraints such that, when executed, the services achieve the desired functionality required by the input service-composition problem.
3. The WSC procedure does not have a complete knowledge of the state of the world; the missing information must be obtained from information-providing services. The WSC procedures execute the information-providing services to obtain the missing information either during the composition process or during the execution of the composition.
4. The WSC procedure does not execute any Web services that have world-altering effects during the composition process.
5. The information returned from the information-providing services is static. *This is the assumption that our work is intended to overcome.*

We now describe a way to take a class of WSC procedures that have the characteristics mentioned above, and modify them to work with volatile information. We start by defining an *unknown* to be any item of information that a WSC procedure needs to obtain to carry out the composition process. For an unknown u, a WSC procedure sends a *query* q_u to the available information-providing Web services that can provide the *value* v_u for u.The value for u is returned by a Web service to the WSC procedure as

```
Procedure General-WSC($P$)
  $S_0 \leftarrow$ create-initial-state($P$); OPEN $\leftarrow \{S_0\}$; ANSWERS $\leftarrow \emptyset$
  loop
    insert all new answers for the pending queries (if any) into ANSWERS
    select a node $S$ from OPEN and remove it
    if solution($S, P,$ ANSWERS) then return extract-solution($S, P$)
    issue queries about zero or more unknowns in $S$ that are not in ANSWERS
    OPEN $\leftarrow$ (OPEN $\setminus \{S\}) \cup$ children-of($S, P$)
```

Fig. 1. The General-WSC procedure is an abstract model of many static-information WSC procedures. It is based on the observation that most existing WSC procedures are trial-and-error search procedures that may try different possible Web service compositions in order to find one that accomplishes the desired task. P is the problem description, and the initial state S_0 is derived from it.

an *answer* for the query q_u. A query issued by a WSC procedure is said to be *pending* if no answers have been received for that query. Otherwise, it is *completed*. A pending query becomes completed if all answers for that query is received.

Our procedural model is the General-WSC procedure shown in Figure 1. This model captures the procedural behavior of most existing service-composition techniques. Examples include [1, 2, 3, 4, 5], and others.

In the General-WSC procedure, each *state* is an abstract representation of a partial solution to the WSC problem. If S is a state, then each child S' of S is obtained by making some kind of refinement to the partial solution represented by S. We assume that whether or not S can be refined to produce S' will depend on some *precondition* *pre*(S, S'), whose value may be $true$ or $false$ depending on the values of some of the unknowns. If *pre*$(S, S') = false$ then the refinement cannot be performed, hence S' is a dead end. But if *pre*$(S, S') = true$ then the refinement can be performed. In the latter case, either S' is a solution to the WSC problem or else it has one or more children of its own. A state is a *terminal* state if it is either a dead end or a solution.

Let S_0 be the initial state of a WSC problem, and let $\langle S_0, S_1, \ldots, S_n\rangle$ be a sequence of states such that each S_{i+1} is a child of S_i and S_n is a terminal node. Then from the above assumptions, it follows that S_n is a solution if and only if

$$\mathit{pre}(S_0, S_1) \wedge \mathit{pre}(S_1, S_2) \wedge \ldots \wedge \mathit{pre}(S_{n-1}, S_n) = true.$$

The variable OPEN is the set of all states that the WSC procedure has generated but has not yet been able to examine. The variable ANSWERS is the set of all answers that have been returned in response to a WSC procedure's pending queries; i.e.,

$$\text{ANSWERS} = \{(u, v) : \text{ a Web service has returned the value } v \text{ for the unknown } u\}.$$

General-WSC begins with a set called OPEN that contains only the initial state S_0. Within each iteration of the loop, General-WSC does the following:

- It updates ANSWERS to include any answers that have been returned in response to its queries.
- It selects $S \in$ OPEN to work on next. Which node is selected depends on the particular WSC procedure. For example, in both the Golog-based [1] and SHOP2-based [2] approaches, the search is performed in a depth-first manner. The PKS-based approach reported in [4] can perform either depth-first or breadth-first search.
- It checks whether or not S constitutes a solution (i.e., a composition that achieves the goals of the current WSC problem). In the pseudocode of Figure 1, this check is represented by the solution subroutine. The definition of the solution subroutine depends on the particular instance of General-WSC. For example, in [1,2,3], solution checks whether or not the sequence of world-altering services can really be executed given the information collected from the information-providing services during the composition process. In the PKS-based approach of [4], the definition of solution includes (1) checking for the correctness and consistency of the knowledge-level databases that PKS maintains, and (2) checking for whether the current solution achieves the goals of the current WSC problem.
- If S is not a solution, then the procedure has an option to issue queries about the unknowns that appear in S. Then it generates the successors of S, and inserts them into the OPEN set. The children-of subroutine is responsible for this operation, and again the details depend on the particular WSC procedure. In [1], children-of a state is defined through the Trans rules described in that work. A successor state generated by those rules specify the next Golog program to be considered by the composition procedure as well as the current partial composition generated so far. In HTN plannning based approaches as in [2, 3], successor states are computed via task-decomposition techniques.

3 Dealing with Volatile Information

The previous section dealt with *static-information* WSC procedures, i.e., WSC procedures for the case where the values of the unknowns will never change. We now consider *volatile-information* WSC procedures, i.e., WSC procedures for the case where values of the unknowns may change over time.

Figure 2 illustrates the life cycle for the value of an unknown u. Suppose a WSC procedure issued a query q_u to a Web service W at time $t = t_{issue}(q_u)$, asking for the value of u. The answer for this query will arrive at time $t_{return}(q_u) = t_{issue}(q_u) + t_{lag}(q_u)$, where the *lag time* $t_{lag}(q_u)$ includes both the time the information-providing service takes to process the query q_u and the time delay due to network traffic.

In addition to the lag times of queries, we also need to consider (1) the time needed to compute a precondition *pre*(S, S'), and the time needed to perform the refinement *refine*(S, S') that takes us from the state S to the state S'. Note that if *pre*$(S, S') = false$, then the time to perform *refine*(S, S') is zero. If *pre* and *refine* refer to unknowns whose values are not currently known, then computing them may require sending queries to Web services, thereby incurring some lag times. We assume that except for those lag times, the time needed to compute *pre* and *refine* is negligible.

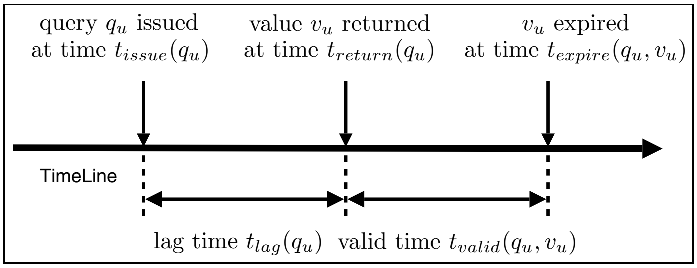
query q_u issued at time $t_{issue}(q_u)$
value v_u returned at time $t_{return}(q_u)$
v_u expired at time $t_{expire}(q_u, v_u)$
TimeLine
lag time $t_{lag}(q_u)$
valid time $t_{valid}(q_u, v_u)$

Fig. 2. A typical execution of an information-providing service. Above, $t_{issue}(q_u)$ is the time that a WSC procedure issues a query to a Web service for the value of an unknown u. $t_{return}(q_u)$ is the time at which the value of u is received, and $t_{expire}(q_u, v_u)$ is the time point after which that value is no longer guaranteed to be valid.

Suppose the answer for q_u specified the value v_u for u. Associated with the answer is a *valid time* $t_{valid}(q_u, v_u)$, i.e., the amount of time that the answer is guaranteed to be valid.[1] This means that the value of the unknown u is guaranteed to be v_u between the times $t_{return}(q_u)$ and $t_{expire}(q_u, v_u) = t_{return}(q_u)+t_{valid}(q_u, v_u)$. At $t_{expire}(q_u, v_u)$, the value v_u *expires*; i.e., u's value is no longer guaranteed to be v_u after the time $t_{expire}(q_u, v_u)$.

Since the values of the unknowns change over time, the correctness of a solution composition returned a volatile-information WSC procedure depends on the values gathered during the composition time. In order to guarantee that the returned composition will be executed correctly on the Web, we will define a solution composition to be T-*correct* if it is guaranteed to remain correct for at least some time T after a WSC procedure returns that solution. In order to provide such a guarantee, we assume that a value obtained for an unknown u will remain valid for at least time T.

A static-information WSC procedure is said to be *sound* if whenever it returns a solution to a WSC problem, the solution is a correct one. By analogy, we will say that a volatile-information WSC procedure is T-*sound* if whenever it returns a solution, the solution is T-correct.

In the following subsections, we introduce two approaches for taking static-information WSC procedures and translating them into volatile-information WSC procedures. For both of them, if the original WSC procedure is sound, the translated procedure will be T-sound.

[1] Some WSC procedures provide a valid time explicitly. For example, hotel rooms can usually be held without charge until 6pm on the night of arrival; and the web site at our university's concert hall will hold seating selections for several minutes (with a countdown timer showing how much time is left). However, our approach does not actually need a valid time to be given explicitly, as long as there is a mechanism to inform the WSC procedure immediately after an expiration has occurred.

However, in that case, the WSC procedure can no longer guarantee how long the solution will remain valid after it is returned, because expirations may occur anytime after the solution is returned.

3.1 The Black-Box Approach

[9] investigated how to generate plans in the presence of incomplete and volatile information. The authors provided a query management strategy that could be wrapped around most automated-planning systems, to manage their queries to external information sources.

Our black-box approach is a modified version of the approach described in [9]. The modifications are: (1) replace the planner with a WSC algorithm, (2) replace the information sources with information-providing Web services, and (3) modify the strategy to pretend that each unknown u's expiration time is $t_{expire}(q_u, v_u) - T$ rather than $t_{expire}(q_u, v_u)$. The latter modification is necessary to ensure that the solution returned by the WSC procedure is T-correct.

[9] also described two query-management strategies that we can use with the black-box approach:

- In the *eager* strategy, when the information collected from external information sources is expired, the query-management strategy immediately re-issues the relevant query or queries and suspends execution of the underlying WSC procedure until the answers come back.
- In the *lazy* strategy, the query-management strategy does not immediately reissue new queries about the expired information. Instead, it assumes that such information is still valid and continues with the composition process until the underlying WSC procedure generates a solution. At that point, the lazy strategy re-issues queries about all expired information that that solution depends on, and suspends execution of the WSC procedure until all of the answers is received.

If the same answers are received for the re-issued queries as before, these strategies restart the WSC procedure from where it left off. With the lazy strategy, this means the procedure immediately returns the solution and exits. Otherwise, the strategies backtrack the WSC procedure to the first point where it made a decision that depends on an unknown whose value has changed, and restarts the procedure from that point.

The following theorem establishes the correctness of the black-box approach:

Theorem 1. *Let A be a WSC procedure that is an instance of* General-WSC*, and let A^B be the modified version of A produced by the black-box approach. If A is sound, then A^B is T-sound.*

For a detailed discussion and analysis on the black-box approach, please see [9].

3.2 The Gray-Box Approach

Although the black-box approach described in the previous section is a simple and a general technique to modify WSC procedures to deal with volatile information, it has one drawback: it does not consider the internal operations of the underlying WSC procedures, and therefore, it may not perform very efficiently in some WSC problems. In this section, we describe another technique, called the *gray-box* approach, that takes into account the internals of WSC procedures that are instances of General-WSC in order to generalize them to deal with volatile information.

```
Procedure VI-General-WSC($P, T$)
  $S_0 \leftarrow$ **create-initial-state**($P$); OPEN $\leftarrow \{S_0\}$; ANSWERS $\leftarrow \emptyset$

  loop
    remove some or all expired answers from ANSWERS
    insert all new answers for the pending queries into ANSWERS
    select a node $S$ from OPEN
    if **solution**($S, P,$ ANSWERS) then
      if $S$ contains no unknowns whose values have expired or will
        expire within time period $T$, then
          return **extract-solution**($S, P$)
      else
        remove zero or more values from ANSWERS that have expired or
          will expire within time period $T$, and re-issue queries about them
        OPEN $\leftarrow$ OPEN $\cup \{S\}$
    else
      issue queries about zero or more unknowns in $S$ that are not in ANSWERS
      OPEN $\leftarrow$ (OPEN $\setminus \{S\}) \cup$ **children-of**($S, P$)
```

Fig. 3. The VI-General-WSC procedure generalizes the General-WSC to deal with volatile information. It returns a solution to the WSC problem that will remain correct for at least T amount of time after the solution is returned.

The gray-box approach is based on a modified version of the General-WSC procedure, called VI-General-WSC, that works with volatile information. This procedure is shown in Figure 3. In this approach, we take an instance of the abstract General-WSC service-composition procedure, and translate it into the corresponding instance of VI-General-WSC.

Like General-WSC, VI-General-WSC performs a search in the space of states, but it also keeps track of the *expired* values for the unknowns for which it issued queries previously, and maintains the ANSWERS set accordingly. At each iteration, a state S in OPEN is *active*, if for every unknown u that appears in S we have $(u, v) \in$ ANSWERS, where v is the value of u in S. In other words, a state in OPEN is active at a particular iteration of VI-General-WSC, if all of the information that it depends on is valid at that iteration. Otherwise, S is *inactive*. As an example, in Figure 4, the solid squares are active states and the dashed squares are inactive ones.

The following theorem establishes the correctness of the gray-box approach:

Theorem 2. *Let A be a WSC procedure that is an instance of* General-WSC*, and let A^G be the modified version of A produced by the gray-box approach. If A is sound, then A^G is T-sound.*

This theorem holds because (1) given a set of unknowns and possible values for them, both A and A^G have the same search traces, and (2) A^G terminates only when the solution satisfies the **solution** function and the values the solution depends on remain valid for time T. Therefore, the solution is T-correct only if A is sound.

Earlier, for the black-box approach, we defined two query-management strategies: the *eager* and *lazy* strategies. In the gray-box approach, since we have some control

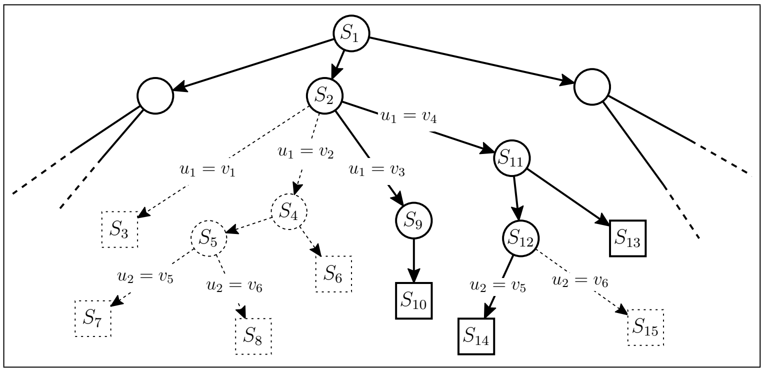

Fig. 4. An example snapshot of a VI-General-WSC search space. There are three unknowns, and their current values are $u_1 = v_3$, $u_1 = v_4$, and $u_2 = v_5$. The squares represent the states in the open list, and the circles represent the states that have already been visited. The label on each edge (S_i, S_j) gives a value $u_h = v_k$ that the refinement *refine*(S_i, S_j) depends on. For example, *refine*(S_{12}, S_{14}) only works if $u_2 = v_5$, and the state S_{14} is a valid refinement of S_1 only if both $u_1 = v_4$ and $u_2 = v_5$. The solid squares denote active states; these represent valid refinements of S_1. The dashed squares denote inactive states: these once were valid refinements of S_1, but they are not currently valid because some of the information they depend on has expired.

over the way underlying WSC procedures perform their search, our query-management strategies can be more sophisticated. For example, here are two query management strategies for use with the gray-box approach:

- The *active-only strategy* selects the first active state from the OPEN set, if there exists any. If there is no active state in the OPEN set, then the composition process stops until some states become active again. In this case, when an answer for a query expires, we immediately re-issue that query.
- The *active-inactive strategy* first attempts to select an active node, if there are active nodes in the OPEN list. If not, it attempts to select an inactive node, assuming that the values for the unknowns that this selection depends on will become valid at some point later in the composition process. In this case, we do not reissue a query after its value is expired; instead, we treat the expired values as if they are not expired. When we get to a goal state, we reissue all the queries for all expired values that some goal state in the OPEN set depends on.

4 Implementation and Experimental Evaluation

In our experiments, we used both the black-box and gray-box techniques to generate volatile-information WSC procedures. In particular, we used the static-information WSC procedure described in [2], which is an instance of the abstract General-WSC procedure. This WSC procedure is based on a translation of OWL-S process models into HTN methods and operators for use within the SHOP2 planning system [7].

In our experiments, we assumed that this translation process had already been carried out, hence we started directly with the SHOP2 methods and operators. We implemented the following four volatile-information WSC procedures:

- Eager and Lazy: black-box translations of the static-information WSC procedure using the eager and the lazy strategies, respectively.
- Active-Only and Active-Inactive: gray-box translations of the static-information WSC procedure using the active-only and the active-inactive strategies, respectively.

For our experiments, we used two service-composition scenarios. The first is the Delivery-Company application described in [3]. In this domain, a delivery company is trying to arrange the shipment of a number of packages by coordinating its several local branches. The company needs to query Web services to gather information from its branch offices about the locations and the availability of vehicles (i.e., trucks and planes) and the status of packages. The goal is to generate a sequence of commands to send as Web service calls to the vehicle controllers, such that the execution of these commands will route all of the packages to their final destinations.

Our second service-composition scenario involves a simplified model for grid- and utility-based computing [10]. In our scenario, there are a number of Grid Services for reserving computing resources owned by several different companies on the Web. Some of them are information-providing grid services giving the current workload, memory usage, software license, etc. The WSC procedure's goals are to figure out which computing resources to use for a given computing task, and to generate a composite Grid Service that actually makes the reservation once it is executed. Since the workload and the memory usage of the machines keep changing, it is necessary for the WSC procedure to deal with the change of information during composition.

We randomly created 7 delivery-company problems and 8 grid-computing problems. Then, in the description of each problem, we randomly inserted n number of unknown symbols, for $n = 1, \ldots, 9$. For each number of unknowns, we ran each problem 50 times and averaged the running times. Every time a query was issued, we generated the lag time for that query and the valid time for the answer by choosing numbers at random from the time interval $0.5 \leq t \leq 2.5$ seconds.

The results are shown in Figures 5 and 6 on Delivery-Company and Utility-Computing problems using an Intel Xeon 2.6GHz CPU with 1GB memory. Each data point is an average of 350 and 400 runs, respectively. Missing data points correspond to experiments where one or more of the runs went for longer than 30 minutes.

In these experiments, the two WSC procedures produced by the gray-box approach (the Active-Only and Active-Inactive procedures) performed much better than the two WSC procedures produced by the black-box approach (the Eager and Lazy procedures). This occurred because the former were able to explore alternative compositions for a problem while awaiting responses from the information-providing services. The improvement in running time was roughly exponential. For example, with 7 unknowns Active-Inactive took roughly 1/50 the time required by the Lazy procedure.

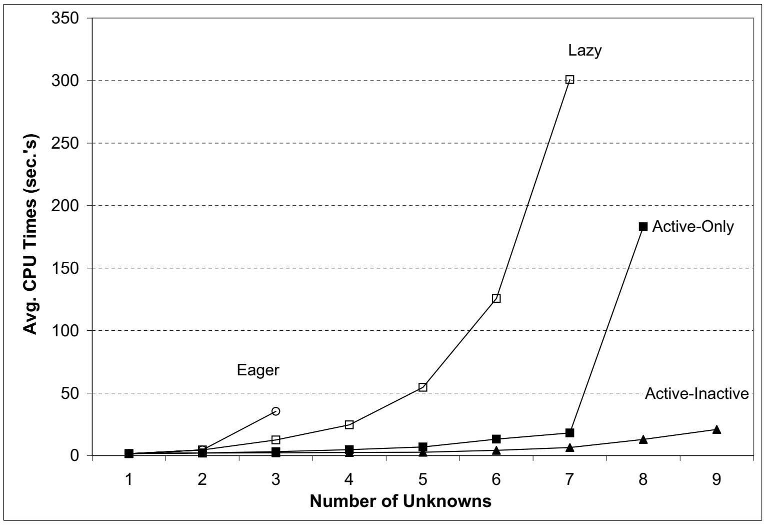

Fig. 5. Average running times of our algorithms on Delivery-Company problems, as a function of the number of unknowns

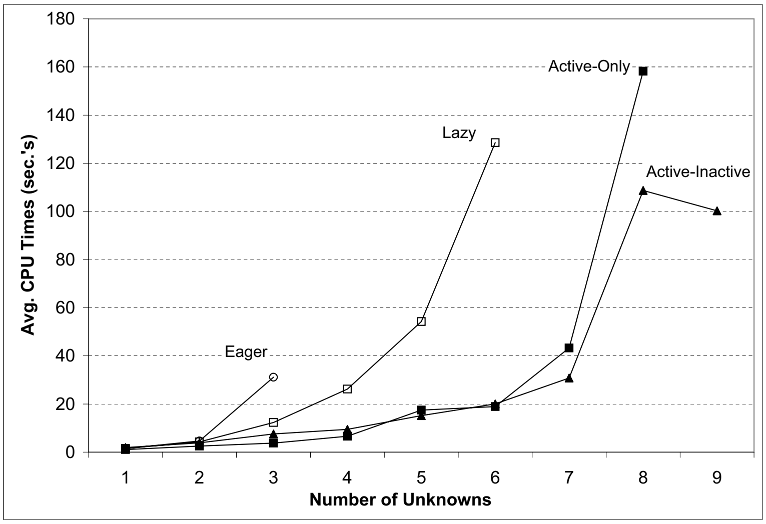

Fig. 6. Average running times of our algorithms on Utility-Computing problems, as a function of the number of unknowns

In addition, the Active-Inactive procedure performed much better than the *Active-Only* procedure.[2] The reason is that in the case when there are no active nodes

[2] Analogously, Lazy performed much better than Eager. This confirms the results reported in [9].

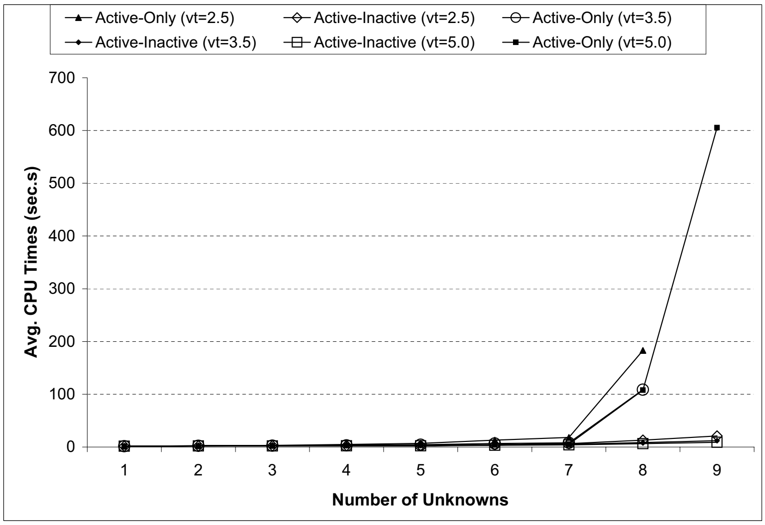

Fig. 7. Average running times of our algorithms on Delivery-Company problems with varying number of unknowns and valid times for the answers of queries. In each case, "vt" denotes the upper bound for the valid times used in the experiments.

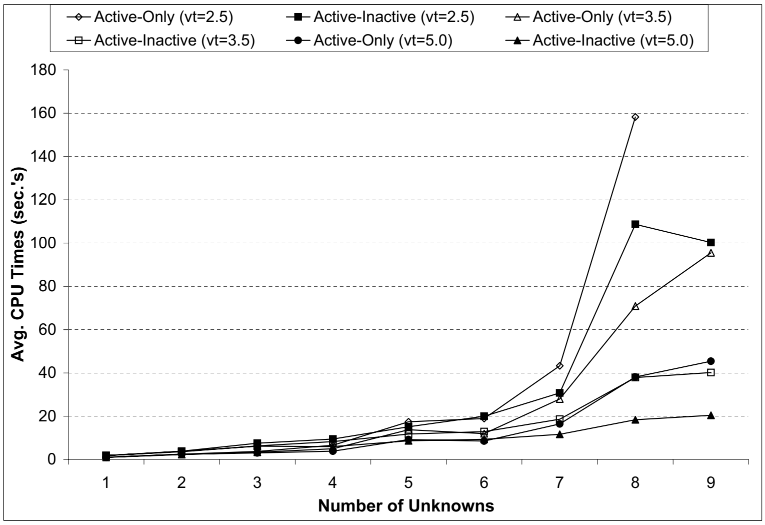

Fig. 8. Average running times of our algorithms on Utility-Computing problems, with varying number of unknowns and valid times for the answers of queries. In each case, "vt" denotes the upper bound for the valid times used in the experiments.

(i.e., when every state in the OPEN set depends on an unknown whose value has expired), Active-Only suspends execution while waiting for responses from Web services, whereas Active-Inactive keeps working: it expands the inactive nodes assuming the responses they depend on may become valid again at some point in the future. This enables Active-Inactive to explore many more alternative compositions than Active-Only in the same amount of time.

Note that the running times of our WSC procedures grow exponentially with the number of unknowns. The reason is that it is getting harder to get all required valid values simultaneously as the number of unknowns increases. In order to further investigate the effect of valid times on the behavior of our WSC procedures, we also did another set of experiments in which we varied the upper bounds for the valid times of the answers for our WSC procedures. In these experiments, we used the same experimental scenarios described above with $2.5, 3, 5$ and 5.0 seconds as upper bounds for the valid times in our random simulation.

Figures 7 and 8 show the results of these experiments on the Delivey-Company and Utility-Computing problems, respectively. As shown in these figures, the performances of our procedures increase dramatically with the increasing valid times for the answers of their queries. The Active-Only procedure was able to solve the problems with 9 unknowns, which it was not able to solve before. The Active-Inactive procedure outperformed Active-Only in all cases. Finally, the performance of Active-Inactive increased expoenentially with the increasing valid times. This is because, when the values obtained for the unknowns do not expire very quickly as in our first set of experiments, Active-Inactive quickly finds a solution as it expands inactive states as well as active ones and returns it before any value that that solution depends on expires.

5 Related Work

In addition to the service-composition techniques [1, 2, 3, 4] described earlier, another WSC procedure that fits into our framework is a technique based on an *estimated-regression* planner called Optop [5]. As an instance of the General-WSC procedure, a state is a *situation* in Optop, which is essentially a partial plan. The solution function checks whether the current situation satisfies the conjunction of the goal literals given to the planner as input, and the children-of function computes a regression-match graph and returns the successors of the current situation.

[11] is another WSC approach that also fits into our framework. It is based on a partial-order planner that uses STRIPS-style services translated from DAML-S descriptions of atomic services to compose a plan. As an instance of the General-WSC, a state is a partial-order plan; the solution function checks if there is any unsatisfied subgoal in a plan, and the children-of function generates child nodes by either instantiating operators or using external inputs or preconditions to satisfy the subgoals. By using our approach, the extended procedure might obtain information about the conditions of the subgoals though Web services during planning.

In [12] and [13], a planning technique based on the "Planning as Model Checking" paradigm is described for the automated composition of Web services. The BPEL4WS process models was first translated into state transition systems that describe the dynamic interactions with external services. Given these state-transition systems, the planning algorithm, using symbolic model checking techniques, returns an executable process rather than a linear sequence of actions. It is not immediately clear to us if this approach fits into the trial-and-error framework that our approaches are based on.

6 Conclusions and Future Work

In this paper, we have described two approaches for taking WSC procedures designed to work in static-information environments, and modifying them to work correctly in volatile-information environments.

The black-box approach requires no knowledge of the internal operation of the original WSC procedure. It puts a wrapper around the procedure to deal with the volatile information.

The gray-box approach requires some knowledge of the original WSC procedure, but only partial knowledge: it requires knowing that the original procedure is an instance of our General-WSC. The gray-box approach works by inserting some additional bookkeeping operations at various points in the instances of General-WSC.

Our experimental results show that despite the simplicity of these modifications, the resulting volatile-information WSC procedures can perform much better than the ones produced by the black-box approach. This is because the modifications enable the volatile-information WSC procedure to explore alternative Web service compositions while waiting for its queries to be answered.

This paper is just a first step in the development of WSC procedures for volatile-information environments. There are several important topics for future work:

- There are situations in which some of the valid times are so short that the WSC procedure cannot finish its task due to an overwhelmingly large number of expirations. Furthermore, there are situations in which the WSC procedure can never get hold of valid values of some of the unknowns simultaneously, and thus it is impossible to return a valid solution. We would like to determine what kinds of conditions are sufficient to guarantee that our procedure will terminate with a solution.
- Like most of the previous work on WSC procedures, we have assumed that the WSC procedure does not execute any Web services that have world-altering effects during the composition process—just the information-providing services. We intend to generalize our work to accommodate the execution of services that have information-providing effects, world-altering effects, or both during service composition.
- Even more generally, we are interested in allowing the possibility of interleaving composition and execution—e.g., to allow the WSC procedure to execute a portion of the composition before generating the rest of the composition.
- We believe the gray-box approach can be made even more efficient by extending it to make use of knowledge of what the search space looks like, and what the solutions should look like.

Acknowledgment. This work was supported in part by NSF grant IIS0412812 and AFOSR grant FA95500510298. The opinions expressed in this paper are those of authors and do not necessarily reflect the opinions of the funders.

References

[1] McIlraith, S., Son, T.: Adapting Golog for composition of semantic web services. In: KR-2002, Toulouse, France (2002)

[2] Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP2. Journal of Web Semantics **1** (2004) 377–396

[3] Kuter, U., Sirin, E., Nau, D., Parsia, B., Hendler, J.: Information gathering during planning for web services composition. In: ISWC-2004. (2004)

[4] Martinez, E., Lespérance, Y.: Web service composition as a planning task: Experiments using knowledge-based planning. In: ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services. (2004)

[5] McDermott, D.: Estimated-regression planning for interactions with web services. In: AIPS. (2002)

[6] OWL Services Coalition: OWL-S: Semantic markup for web services (2004) OWL-S White Paper http://www.daml.org/services/owl-s/1.1/owl-s.pdf.

[7] Nau, D., Au, T.C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., Yaman, F.: SHOP2: An HTN planning system. JAIR **20** (2003) 379–404

[8] Petrick, R.P.A., Bacchus, F.: A knowledge-based approach to planning with incomplete information and sensing. In: AIPS. (2002)

[9] Au, T.C., Nau, D., Subrahmanian, V.: Utilizing volatile external information during planning. In: ECAI. (2004)

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

[11] Sheshagiri, M., desJardins, M., Finin, T.: A planner for composing services described in daml-s. In: AAMAS Workshop on Web Services and Agent-based Engineering. (2003)

[12] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: AIMSA. (2004)

[13] Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: ISWC. (2004)