Abstract

Web service composition (WSC) problems involve using domain knowledge about the underlying problem domains during the composition process. Existing research on Web service composition procedures has generally assumed that this domain knowledge is encoded as a single ontology that must be provided as an input to a composition procedure. In composition problems that span over knowledge across multiple ontologies that are connected via concept inheritance/extension, the users must examine the available ontologies and services in order to decide for the extent of the domain knowledge and the set of services relevant to their composition problems in hand.

In this paper, we describe an automated way to generate solutions for service composition problems that span over multiple ontologies that have similar but not exactly the same structure. Our approach is based on a Hierarchical Task Network (HTN) planning model and extends the HTN-DL framework. This HTN planning model allows ontological problem decomposition in order to evaluate structural properties that span across multiple ontologies that are relevant to an input composition problem. We present a demonstration of the advantages of this approach.

1. Introduction

The Semantic Web is an extension of the World Wide Web and provides a common framework to allow people to share and reuse information on the Internet. A new layer of machine understandable data is added in order to allow computers to participate on the web. A Web service is defined as a software system designed to support machine to machine interaction over network. Many efforts have been made to integrate these two areas and support greater automation of web service composition. Existing research on semantic Web services has focused on Web service composition (WSC) procedures, i.e., procedures for finding a composition of services to accomplish a given task or functionality. The well known WSC procedures described in [12, 9, 17, 10] have generally assumed that a composition problem involves a single domain ontology and all of the services required to solve that problem. This is not very realistic especially for composition problems, such as the one above, that spans over multiple ontologies with only one copy of the standard top-level domain knowledge ontology.

In this paper, we describe an automated way to do Web service composition that spans over multiple ontologies and Web services described for those ontologies. We describe an extension of the HTN-DL framework described in [16] that combines Hierarchical Task Networks and Description Logics. Our extension involves the capability of describing task-decomposition rules based on boolean queries of terminological component (a.k.a. TBox in Description Logic [6]) in HTNs. This allows us to use terminological concepts and definitions during a planning phase in which the planner decomposes the given composition problem into smaller problems, and while doing so, it evaluates the struc-
tural properties of the current domain ontology and the relevant other ontologies, and identifies services to be used in those smaller problems. Once the procedure generates service-composition problem that we know the domain ontology and the necessary Web services, it attempts to solve the problem using the existing techniques in HTN-DL. Our preliminary implementation and evaluation demonstrate the advantages of our approach.

2. Background: HTN Planning and the HTN-DL Framework

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). HTN planning keeps a state of the world and uses actions to represent state transitions as classical planning does. Also as classical planning, HTN planners operates using the Closed World Assumption, i.e., the assumption that if a planner cannot infer a fact from its state then that fact is assumed to be false.

HTN planners differ from classical planners, however, in what they plan for and how they plan for it. The description of a planning problem contains an initial knowledge base as in classical AI planning [4]. 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 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 will backtrack and try other methods. For a detailed discussion on HTN planning, see [4, 13].

Although HTN planning has been proved promising for Web service composition [17, 9], previous work has some limitations in their use of HTN planning directly for web services described in OWL-S because of the following reasons:

- In HTN planning there is no way to associate preferences about possible decomposition, such as mapping tasks with taxonomies.
- HTN planning has restrictions on primitive and nonprimitive tasks such that it is not possible to say an atomic service and a composite service achieve the same goal. This means we cannot match a task with a method and an operator at the same time, but it is common in the web service domain that atomic and composite services have the same functionalities.
- In HTN planning only one operator can accomplish a primitive task. However, this is not true in web services; there are many different web services that can accomplish the same primitive task.
- The precondition and effects in OWL-S description are expressed in some formalisms (e.g. SWRL and KIF) that makes the open world assumption. This is not supported in HTN planning.

HTN-DL [16] is a planning formalism that combines HTN planning with OWL and Description Logic (DL) [6] representation. DL is used to describe actions and states with an expressive knowledge representation language. An HTN-DL task is a concept in a task ontology and that concept represents a service category. Web services belonging to a category defined by a task must provide the same functionality as that task, and HTN-DL determines whether a Web service belongs to a category based on the preconditions and the effects of the services. Figure 1 shows a travel task and three services that are associated with that task.

In HTN-DL, web services are described in OWL-S language. OWL-S partitions a service description into three components: service profile, process model and grounding. Before HTN-DL starts planning, the OWL-S service descriptions are encoded as HTN-DL domains. For each service, the profile description is translated to an element in the task ontology and the process model is translated to a set of methods and operators.

OWL-S process ontology is a quite extensive orchestration language that provides control flow elements such as Perform, Sequence, Any-Order, Split, If-Then-Else, Repeat-While and Repeat-Until. A composite service can

<table>
<thead>
<tr>
<th>Task</th>
<th>Travel (?p, ?a, ?b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Precondition: Person(?p), Location(?a), At(?p, ?a)</td>
<td></td>
</tr>
<tr>
<td>- Effects: At(?p, ?b)</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>Walk (?p, ?a, ?b)</td>
</tr>
<tr>
<td>- Precondition: At(?p, ?a), WalkingDistance(?a, ?b)</td>
<td></td>
</tr>
<tr>
<td>- Effects: At(?p, ?b)</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>Drive (?p, ?a, ?b)</td>
</tr>
<tr>
<td>- Precondition: At(?p, ?a), DrivingDistance(?a, ?b)</td>
<td></td>
</tr>
<tr>
<td>- Effects: At(?p, ?b)</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>Fly (?p, ?a, ?b)</td>
</tr>
<tr>
<td>- Precondition: At(?p, ?a), FlyingDistance(?a, ?b)</td>
<td></td>
</tr>
<tr>
<td>- Effects: At(?p, ?b)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. An example task and three example services that matches to that task.
be built from other processes by using these constructs. HTN-DL translates each OWL-S control construct to an HTN-DL task and a set of HT-DL methods achieving that task. Therefore, HTN-DL can handle control flow elements such as conditional branching or loops as OWL-S does.

HTN-DL translates atomic services in OWL-S descriptions to HTN-DL operators. An operator has input, output, preconditions and effects as an atomic service does. Similarly, composite services in OWL-S are translated to HTN-DL methods. An HTN-DL method is an expression of the form (:method m in out local pre eff tn) where m is the task symbol, in and out are sets of input and output parameters, local is the set of local variables in conditions, pre is the precondition expressions, eff is the effect expressions, and tn is a conditional task network stating that under different conditions we have different ways of decomposition. A task can be matched with operators or methods. If an operator is matched, the planner will use the action and apply effects. If a method is matched, the planner continues decomposition until atomic action is reached. Besides the conditional atoms in preconditions, methods also have conditions in their conditional task network tn.

Figure 2 shows an example of travel domain. The method myTravel can take a person ?p to destination ?b although we have no information about where the person is located. In this method we use a local variable ?a to denote the place ?p is located. In the preconditions the relations between ?p, ?a and ?b are defined, and the knowledge base will evaluate the value of ?a. If the knowledge base fails to find a binding of ?a, the precondition is false and the planner backtracks to previous branching point. If the knowledge base can evaluate one or more possible bindings of ?a, the planner will use the value and continue planning.

The problem of evaluating conditional expressions is directly reducible to DL query answering. The preconditions of operators and conditions in the conditional task network require a boolean answer, i.e. true or false. The preconditions of methods are either boolean or retrieval queries and the knowledge base returns true with possible local variable bindings or false. Answering boolean conjunctive queries can be reduced to knowledge-base consistency checking [7, 8], and answering retrieval queries can be reduced to boolean query answering by substituting the local variables in the query with the individuals in a knowledge base.

In Figure 2, we use the conditional task network to choose a task network to continue planning. That is, if the distance between source and destination are within walking distance, we choose the Walk service; if within driving distance, we choose the Drive service; otherwise we may take a flight to the destination.

There are two kind of queries accepted in HTN-DL conditions, called TypePattern and EdgePattern. TypePattern is to check if an instance belongs to a given class, e.g.

<table>
<thead>
<tr>
<th>Method myTravel</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Input: (?p, ?b)</td>
</tr>
<tr>
<td>- Local: (?a) - Precondition: Person(?p), Location(?a), At(?p, ?a)</td>
</tr>
<tr>
<td>- Effects: At(?p, ?b)</td>
</tr>
<tr>
<td>- Conditional Task Network:</td>
</tr>
<tr>
<td>- if (isWalkingDistance(?a, ?b)) then Walk(?p, ?a, ?b)</td>
</tr>
<tr>
<td>- if (isDrivingDistance(?a, ?b)) then Drive(?p, ?a, ?b)</td>
</tr>
<tr>
<td>- else Fly(?p, ?a, ?b)</td>
</tr>
</tbody>
</table>

| Figure 2. An example task and three example services that matches to that task. |

City(?c) to check if ?c belongs to class City. EdgePattern describes two instances have a relation of a given property, e.g. locatedIn(?a, ?c) means airport ?a is at city ?c. Person(?p) and Location(?a) belongs to TypePattern while At(?p, ?a), isWalkingDistance(?a, ?b) and isDrivingDistance(?a, ?b) are of EdgePattern.

3. Our Model

We use the same definitions for constant symbols, variable symbols, logical expressions, knowledge-base assertions, tasks, and task networks as in [16].

We assume the existence of a finite set B of concept definitions, called scoping concepts, of the form (ns:name) where ns is the namespace of the original ontology on the Web that the concept named name is defined in. Intuitively, the scoping concepts in B provide the information about the possible ontological connections/links a WSC procedure could use during the composition process.

Note that importing all of the ontologies into the local ontology of a WSC procedure would not solve the problem, since it may introduce inconsistencies between the ontological concepts. This is the main motivation behind the existing research on reasoning with distributed ontologies [2, 3, 5].

We define a finite set of special task symbols, called problem task symbols. A problem task is an expression of the form (solve task arg1 arg2 ... argk), where task is a unique problem-task symbol, and each argument argi is either a constant symbol or a variable symbol. A problem task is accomplished using the problem-decomposition methods. Formally, a problem-decomposition method is an expression of the form (:pd-method head pre dec), where head describes a problem-task, pre is a logical expression that consists of description logics assertions, and dec is the decomposition, i.e., the conditional task network stating that under different conditions we have different ways of decomposition of the problem task specified by the head of this method. Given a knowledge base K, a problem task t and a problem-decomposition method m, m is applicable to t if
there exists a variable substitution \( \sigma \) such that \( \sigma(\text{head}) = t \) and \( \sigma(\text{pre}) \) is satisfied in \( K \).

Intuitively, a problem-decomposition method is similar to a standard HTN-DL task-decomposition method, with the following differences. First, the preconditions of a problem-decomposition method may include more expressive queries than HTN-DL supports, as we describe below. Second, a subtask of a problem-decomposition method is either a problem-task or a standard HTN-DL task. We assume that a regular HTN-DL task cannot be decomposed into a problem-task in the hierarchical task decomposition process. Finally, a problem-decomposition method does not have any effects.

The aforementioned two patterns, i.e., TypePattern and EdgePattern, that constitute the conditions in HTN-DL methods and operators may be enough for most cases about instances, however, they are not sufficient for choosing among multiple ontologies during the composition process; this is because most of the cases, such choices do not depend on the instances but rather the ontological structure of class/concept definitions. For example, suppose we are planning over the package shipping domain. In some places, urgent packages are sent by air and in other places urgent packages are sent by ground vehicles. We can easily think of countries or regions like Singapore or Hong Kong, which only has one airport so all domestic packages are shipped by ground transportation. Thus, even though we do not know what ontology the user will use in planning, we can check the subsumption. Thus, even though we do not know what ontology all domestic packages are shipped by ground transportation in Singapore or Hong Kong, which only has one airport so all domestic packages are shipped by ground transportation. Thus, even though we do not know what ontology the user will use in planning, we can check the subsumption. Thus, even though we do not know what ontology all domestic packages are shipped by ground transportation.

A precondition of a problem-decomposition method then can be formulated as a query to the knowledge base in the form of subsumption, equivalence, disjoint and complement. For example, we can have TBox queries like \( \text{Lecturer} \sqsubseteq \text{us-edu:Staff} \) or \( \text{City} \sqsubseteq \text{uk:MajorCity} \). Local concepts can also appear on the right hand side and they can be more complex than the example above: e.g., the definition of the parent with three children: \( \text{3KidParent} \equiv =3 \text{hasChild} \).

As another example, suppose we want to find an airport and ship packages by airplane. Suppose we have a city concept in our local domain ontology. If our city concept extends a concept from \( B \) on a major city, there must be an airport and we use intracity transportation. If our city concept extends a concept from \( B \) about small cities, then we go to a nearby major city. The following shows a problem-decomposition method in our formalism that represents the above choice between the ontologies.

<table>
<thead>
<tr>
<th>pd-method gotoAirport (?c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition: (City(?c))</td>
</tr>
<tr>
<td>network:</td>
</tr>
<tr>
<td>( [subClassOf(City, MajorCity)] - [intracity-transport] )</td>
</tr>
<tr>
<td>( [subClassOf(City, MinorCity)] - [intracity-transport] )</td>
</tr>
</tbody>
</table>

We define a service-composition problem to be a tuple of the form \( P = (K, B, X, T, A, M) \), where \( K \) is a (possibly incomplete) initial knowledge base, \( B \) is the finite set of scooping concepts, \( X \) is a set of all possible Web Services that are available during the planning process, \( T \) is a problem task, \( A \) is a set of problem-decomposition methods, and \( M \) is a collection of OWL-S process models.

A solution for a service-composition problem is a sequence of atomic services that, when executed, solves the problem task \( T \).

Note that our model described above allows us to solve service composition problems that span over multiple ontologies and Web services described for those ontologies. In the next section, we describe a way to distinguish between multiple ontologies based on the task we are trying to accomplish.

## 4. High-Level Planning for Service Composition Across Multiple Ontologies

Figure 3 shows the pseudocode of our high-level planning procedure, called Decompose. Decompose takes a service-composition problem \( (K, B, X, T, A, M) \) as an input and successively decomposes the problem tasks into smaller problem tasks using problem-decomposition methods. The problem-decomposition process continues until an OWL-S service is reached to be decomposed next. At that point, Decompose invokes HTN-DL in order to generate
Procedure Decompose\(P = (K, B, X, T, A, M)\)
1. \(OPEN \leftarrow \{(K, T)\}; \pi \leftarrow \emptyset\)
2. loop
3. if \(OPEN = \emptyset\) then return \(\pi\)
4. select the pair \((K, T)\) from \(OPEN\) that has no predecessors and remove it
5. if \(T\) is a problem task then
6. choose a problem-decomposition method \(m\) from \(A\) that is applicable to \(T\)
7. \(OPEN \leftarrow \text{ApplyMethod}(K, m, T, OPEN)\)
8. else if \(T\) is an OWL-S service then
9. \(\pi' \leftarrow \text{HTN-DL}(K, X, T, M)\)
10. if \(\pi' = \text{FAILURE}\) then return \(\text{FAILURE}\)
11. \(\pi \leftarrow \pi \cup \pi'\)

Figure 3. The abstract Decompose procedure for doing high-level for problem decomposition until the problem can be solved.

```plaintext
if ( City \sqsubseteq \text{us:City} )
  \{ moveTruck(?t, ?c1,?c2), pickup(?p, ?t),
       moveTruck(?t, ?c2, ?c3), drop(?p, ?t) \}
else if ( City \sqsubseteq \text{cn:City} )
  \{ moveBike(?b, ?c1,?c2), pickup(?p, ?b),
       moveBike(?b, ?c2, ?c3), drop(?p, ?b) \}

... By using the problem task and their decomposition methods, Decompose allows users to ignore the details or structure of the ontology. They simply design and use their own ontologies for service composition and the planner will check the ontology and choose a correct set of services and generate a solution composition for the users. Note that we are not trying to give meanings to the namespaces; in this case we assume \text{us} and \text{cn} are two well known ontologies and the experts utilize concept subsumption with these two namespaces to distinguish ontologies.

If all the preconditions of the problem-decomposition method \(m\) is satisfied in the current knowledge base, then \(m\) is applicable to \(T\) as we described earlier. In this case, Decompose generates all of the subtasks of \(T\) specified in \(m\) and inserts those subtasks in to the \(OPEN\) list. In Line 7 of the pseudocode shown in Figure 3, the \text{ApplyMethod} subroutine is responsible applying the method \(m\) to \(T\), generating its subtasks, and inserting those subtasks into \(OPEN\). This subroutine ensures that the subtasks of \(T\) that are inserted in \(OPEN\) satisfies certain constraints such as ordering constraints between the subtasks inserted and the other existing tasks already in \(OPEN\).

As an example, consider the package-shipping scenario where Decompose needs to reason with two different ontological concepts for location. Suppose the scoping concepts \(B\) defines a concept as \text{us:City} and another concept as \text{cn:City}. Now, if the user’s local ontology \(L\) given to Decompose defines a \text{City} as a subclass of \text{us:City}, then this means that the user wants Decompose to use a concept about US location. Similarly, if \text{City} is a subclass of \text{cn:City} then the user is likely using a concept of China location. Then we can easily recognize these ontologies by checking their differences. The following TBox query is an example of this:

a composition of atomic services that will achieve the functionality desired by the OWL-S service.

Decompose’s \(OPEN\) set is a (possibly partially) ordered set of pairs of the form \((K, T)\) where \(K\) is the current knowledge base and \(T\) is the current problem task. At each iteration, Decompose first checks the \(OPEN\) set: if \(OPEN\) is the empty set then this means that there are no other problem tasks left to be accomplished, and therefore, the composition \(\pi\) is a solution to the input service-composition problem. Thus, Decompose returns \(\pi\). If \(OPEN\) is not the empty set, then Decompose selects a pair \((K, T)\) from \(OPEN\) that has no predecessors and removes it.

If \(T\) in the selected pair \((K, T)\) is a problem task then Decompose chooses a problem-decomposition method \(m\) from \(A\) for \(T\). The problem-decomposition method effectively distinguishes between the multiple ontologies available in the service-composition problem statement by looking at the class structure in those ontologies and their connections using its TBox preconditions.

As an example, consider the package-shipping scenario where Decompose needs to reason with two different ontological concepts for location. Suppose the scoping concepts \(B\) defines a concept as \text{us:City} and another concept as \text{cn:City}. Now, if the user’s local ontology \(L\) given to Decompose defines a \text{City} as a subclass of \text{us:City}, then this means that the user wants Decompose to use a concept about US location. Similarly, if \text{City} is a subclass of \text{cn:City} then the user is likely using a concept of China location. Then we can easily recognize these ontologies by checking their differences. The following TBox query is an example of this:
plished to solve the input planning problem.

If the currently-selected task \( T \) is not a problem task, then this means that \( T \) is an OWL-S service. In this case, \texttt{Decompose} simply invokes HTN-DL to generate a composition (i.e., a sequence of atomic services) that accomplishes the functionality of \( T \). At this invocation, if there is a composition for \( T \) given the current knowledge base and available services, then HTN-DL returns a composition \( \pi' \) (as shown in Line 9 in Figure 3). Otherwise, HTN-DL returns \texttt{FAILURE}. In the latter case, \texttt{Decompose} immediately returns failure since there is no solution to the input service-composition problem. In the former case, it combines the composition returned by HTN-DL at this invocation with the current partial composition \( \pi \).

\texttt{Decompose} terminates when all of the tasks in \texttt{OPEN} are accomplished successfully; i.e., when \texttt{OPEN} is empty.

5. Implementation

We implemented the approach described above in the HTN-DL system. We extended HTN-DL to allow to perform TBox queries for our problem-decomposition methods. The abstract \texttt{Decompose} procedure described above then can directly be implemented using the existing HTN task-decomposition mechanisms in HTN-DL.

To the best of our knowledge, there are no benchmark domains for web service composition with multiple ontologies. So we designed some experiments in an adaptation of an AI planning domain, called \textit{UM Translog} [1]. In the problem domain, we are sending packages from one city to another by various transportation ways. In the ontology we have location classes \texttt{Region, City, Airport}, package classes \texttt{AirPackage, GroundPackage, UrgentPackage, RegularPackage, MailPackage, BulkyPackage, ValuablePackage} and \texttt{HazardousPackage}. We have vehicle classes \texttt{Airplane, Truck} and \texttt{Motorcycle}. There are also properties describing the relations between classes, for example, \texttt{locatedIn(?package, ?city)}.

In the experiment we made the following assumptions for exposition purpose:

- In Brazil, urgent packages across region are sent by air but in Hong Kong they are all shipped by ground transportation. If \texttt{UrgentPackage} is a subclass of \texttt{br:UrgentPackage} then we ship it by air; if \texttt{UrgentPackage} is a subclass of \texttt{hk:UrgentPackage} then we ship it by ground transportation.
- In Canada all airports are located in the suburbs and not in the city downtown. In Japan, the airports are located in the cities. If \texttt{Airport} is a subclass of \texttt{jp:Airport}, then the airports are located inside cities. If \texttt{Airport} is a subclass of \texttt{ca:Airport}, then the airports are located in the regions, not in cities.
- In the US, intracity transportation is by truck. In China, intracity transportation is by motorcycles because some places are not accessible by trucks. Thus, if \texttt{City} is a subclass of \texttt{us:City}, then intracity transportation is by truck. If \texttt{City} is a subclass of \texttt{cn:City}, we use motorcycles to send packages.

The following is the example problem and the planning domain knowledge we need for solving the problem. We also demonstrate two cases with different ontologies, and we show how different plan are generated.

Suppose we have three regions \( r_1, r_2, r_3 \), five cities \( c_1, c_2, \ldots, c_5 \) and three urgent packages \( p_1, p_2, p_3 \). City \( c_1 \) and \( c_2 \) are in region \( r_1 \), city \( c_3 \) is in region \( r_2 \), and city \( c_4 \) and \( c_5 \) are in region \( r_3 \). Package \( p_1 \) is to be shipped from \( c_1 \) to \( c_4 \), package \( p_2 \) is from \( c_2 \) to \( c_1 \), and package \( p_3 \) is from \( c_3 \) to \( c_5 \).

The HTN decompositions are as follows. The planner first checks if the source and destination cities are the same city, in the same region, or across regions. If they are the same city, the planner applies intracity transportation to deliver that package. If they are in the same region, the planner will send a ground vehicle to ship the package. If they are across regions, the planner checks if \texttt{UrgentPackage} is a subclass of \texttt{br:UrgentPackage} or \texttt{hk:UrgentPackage}. If it is the former case, the package will be sent to a nearest airport, shipped by air, sent to destination city and delivered. Otherwise it will transported by ground vehicles. Figure 4 shows the TBox assertions in the preconditions of our problem-decomposition methods that correspond to these cases.

In the case of air transportation, the planner checks if the
Airport is subsumed by ca: Airport or jp: Airport. In the former case, the airports are located in the region and the package is sent by ground to the nearest airport; otherwise they are located within the cities and intracity transportation is used. For intracity transportation, the planner checks City subsumption. If City is a subclass of us: City then trucks are used to send packages. If City is subsumed by cn: City, then we use motorcycles to deliver packages.

In the case (a) of Figure 4, the planner is to send the package $p_1$ from $c_1$ to $c_4$. In this case, the shipping is across regions, since the UrgentPackage is subclass of br: UrgentPackage, $p_1$ is shipped by air transportation. The airports are regional airports so the planner applies move operator to send $p_1$ from $c_1$ to airport $a_1$, and fly operator to send $p_1$ from airport $a_1$ to $a_3$. Then, the planner uses move operator to send $p_1$ from $a_3$ to airport $c_1$, and deliver $p_1$. The following is the plan generated by our planner:

\[
\begin{align*}
&\text{ship } p_1 \text{ from } c_1 \text{ to } c_4 \\
&\text{move}(p_1, c_1, a_1), \text{fly}(p_1, a_1, a_3), \text{move}(p_1, a_3, c_4), \text{deliver}(p_1) \\
&\text{ship } p_2 \text{ from } c_2 \text{ to } c_1 \\
&\text{move}(p_2, c_2, c_1), \text{deliver}(p_2) \\
&\text{ship } p_3 \text{ from } c_3 \text{ to } c_5 \\
&\text{move}(p_3, c_3, a_2), \text{fly}(p_3, a_2, a_3), \text{move}(p_3, a_3, c_5), \text{deliver}(p_3)
\end{align*}
\]

In the case (b) of Figure 4, since the urgent packages are Hong Kong urgent packages, all shipping will be sent by ground. So we need to send packages to destination by ground transportation and then deliver them. The following is the plan generated by our planner:

\[
\begin{align*}
&\text{ship } p_1 \text{ from } c_1 \text{ to } c_4 \\
&\text{move}(p_1, c_1, c_4), \text{deliver}(p_1) \\
&\text{ship } p_2 \text{ from } c_2 \text{ to } c_1 \\
&\text{move}(p_2, c_2, c_1), \text{deliver}(p_2) \\
&\text{ship } p_3 \text{ from } c_3 \text{ to } c_5 \\
&\text{move}(p_3, c_3, c_5), \text{deliver}(p_3)
\end{align*}
\]

From above we can see that different ontologies make different choices at branching points and result in different plans. These choices between ontologies occur in the problem-task decomposition methods in the upper levels of the decomposition process. Due to our assumption stated previously that a standard HTN-DL task cannot be decomposed into a problem task, the choices among ontologies are made sometimes even before the planner actually generates the instance bindings. If only instance-based queries are allowed in the choice points as in HTN-DL and in most other systems, the system would not be able to make such choices between ontologies because, in most cases, instances are generated after some services are executed.

6. Related Work

One of the first approaches for Web Service Composition is the Golog-based technique of [12]. This work is based on the notion of generic procedures and customizing user constraints and extends the Golog language to enable programs that are generic, customizable, and usable in the context of the Web. They also successfully augmented a ConGolog interpreter that combines online execution of sensing actions with offline simulation of world altering services.

Other existing approaches that are similar to our framework are the HTN-planning based techniques described in [17] and [9]. Sirin [17] presented a technique to translate process models and service descriptions written in OWL-S into the problem and domain description language of the HTN planner SHOP2 [13], and then, shows how to solve service-composition problems using SHOP2 based on that translation methodology. Kuter described an HTN planning algorithm, called ENQUIRER [9], which extends the work in [17] by performing a continuous search for compositions while information-providing services are being executed for gathering information during plan time.

Other planning-based approaches for composing Web Services include techniques based on symbolic-model checking [15, 18], knowledge-based planning [10, 14], and regression-based heuristic planning [11]. A common characteristic of these approaches is that they produce conditional plans (i.e., compositions) to accommodate certain contingencies that may arise during the execution of the plan. In the case of symbolic-model checking techniques, the plans can also specify iterative behaviors. In this work, we did not aim to generate conditional plans, but we acknowledge that it should not be hard to find cases where generating conditional plans is useful. We are currently investigating such examples and possible ways of extending our work to accommodate those cases.

Working with multiple ontologies is also a main focus in this paper. OWL has limited support for ontology modularity. One ontology can only import the other ontologies in its entirety and share the same and global semantics. The work in [2, 3] extended Description Logic formalism to Distributed Description Logic and dealt with mappings between domains by using bridge rules. Bridge rules simulated concept subsumption with a special type of roles but they were not exactly concept subsumption. In [5] E-connection helps ontology developers partially import concepts from other ontologies without taking the whole foreign ontologies. E-connection also breaks large ontologies into smaller pieces to provide more effective knowledge
modeling and reasoning.

7. Conclusions

Previous work on Web service composition has generally assumed that the experts write domain knowledge on the assumption that the users will use the same ontology as they do. However, in some cases there are ontologies with similar but not exactly the same class structure, and different composition problems may require to choose an appropriate one to generate plans depending on the problem specifications. In this work we have described a way for doing planning for web service composition in the domains whose ontologies have similar but not exactly the same structure. We presented an HTN planning procedure that can evaluate TBox query patterns in the HTN method preconditions. This procedure allows us to decompose a service-composition problem that requires the use of multiple ontologies into smaller composition problems, and while doing so, it evaluates the structural properties of the current domain ontology and the relevant other ontologies, and identifies services to be used in those smaller problems.

We are currently working on extending our approach to take the advantage of modular ontology techniques such as bridge rules or E-connection without importing the entire foreign ontologies. In the long term, we are planning to generalize our technique described in this paper to do planning under more complex interactions between multiple ontologies that exhibit conflicts and inconsistencies when used together in a service-composition problem.

Acknowledgements. We thank to our anonymous reviewers for their helpful comments in preparation of the final version of this paper.

This work was completed with funding from Fujitsu Laboratories of America – College Park, Lockheed Martin Advanced Technology Laboratory, NTT Corp., Kevric Corp., SAIC, National Science Foundation, National Geospatial-Intelligence Agency, Northrop Grumman Electronic Systems, DARPA, US Army Research Laboratory, NIST, and other DoD sources.

References