



HTN planning for Web Service composition using SHOP2

Evren Sirin^{a,*}, Bijan Parsia^b, Dan Wu^a, James Hendler^a, Dana Nau^a

^a Computer Science Department, University of Maryland, College Park, MD 20742, USA

^b MIND Lab, University of Maryland, 8400 Baltimore Avenue, College Park, MD 20742, USA

Received 11 March 2003; received in revised form 29 June 2004; accepted 29 June 2004

Abstract

Automated composition of Web Services can be achieved by using AI planning techniques. Hierarchical Task Network (HTN) planning is especially well-suited for this task. In this paper, we describe how HTN planning system SHOP2 can be used with OWL-S Web Service descriptions. We provide a sound and complete algorithm to translate OWL-S service descriptions to a SHOP2 domain. We prove the correctness of the algorithm by showing the correspondence to the situation calculus semantics of OWL-S. We implemented a system that plans over sets of OWL-S descriptions using SHOP2 and then executes the resulting plans over the Web. The system is also capable of executing information-providing Web Services during the planning process. We discuss the challenges and difficulties of using planning in the information-rich and human-oriented context of Web Services.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Web Services; Web Service composition; OWL-S; HTN planning; SHOP2

1. Introduction

As Web Services—that is, programs and devices accessible via standard Web protocols—proliferate, it becomes more difficult to find the specific service that can perform the task at hand. It becomes even more difficult when there is no single service capable of performing that task, but there are combinations of ex-

isting services that could. Sufficiently rich, machine-readable descriptions of Web Services would allow the creation of novel, compound Web Services with little or no direct human intervention. Semantic Web languages, such as the Web Ontology Language (OWL) [1] or its predecessor DAML + OIL [2], provide the foundations for such sufficiently rich descriptions.

The OWL-services language [3] (OWL-S),¹ is a set of ontologies for describing the properties and capabilities of Web Services. The OWL-S is designed to support effective automation of various Web Services

* Corresponding author. Tel.: +1 301 405 2696;
fax: +1 301 405 6707.

E-mail addresses: evren@cs.umd.edu (E. Sirin),
bparsia@isr.umd.edu (B. Parsia), dandan@cs.umd.edu (D. Wu),
hendler@cs.umd.edu (J. Hendler), nau@cs.umd.edu (D. Nau).

¹ The previous version of OWL-S was called DAML-S and was based on DAML + OIL.

related activities including service discovery, composition, execution, and monitoring.

For our work, we are motivated by issues related to automated Web Service composition. The OWL-S process ontology provides a vocabulary for describing the composition of Web Services. This ontology uses an “action” or “process” metaphor for describing Web Service behavior—that is, primitive and complex actions with preconditions and effects.

Given a representation of services as actions, we can exploit AI planning techniques for automatic service composition by treating service composition as a planning problem. Ideally, given a user’s objective and a set of Web Services, a planner would find a collection of Web Services requests that achieves the objective. We believe that HTN planning is especially promising for this purpose, because the concept of task decomposition in HTN planning is very similar to the concept of composite process decomposition in OWL-S process ontology. In this paper, we explore how to use the SHOP2 HTN planning system [4,5] to do automatic composition of OWL-S Web Services.

In Section 2, we describe a sample scenario for our research. In Section 3, we give the background knowledge about OWL-S process ontology and SHOP2. In Section 5, we present our approach for automatic Web Services composition. In Section 4, we describe why we think HTN planning is suitable for Web Service composition. In Section 6, we describe the implementation. In Section 7, we discuss the challenges and difficulties of using planning for composing Web Services on Semantic Web. In Section 8, we summarize some related work. And finally, in Section 9, we conclude our work and present some future research directions. Throughout this paper, we use the example we described in Section 2 to illustrate our approach. But our work is designed to be domain-independent and is not restricted to this example.

2. Motivating example

The example we describe here is based loosely on a scenario described in the Scientific American article about the Semantic Web [6]. Suppose Bill and Joan’s mother goes to her physician complaining of pain and tingling in her legs and the physician proposes the fol-

lowing sequence of activities:

- a prescription for Relafen, an anti-inflammatory drug;
- an MRI scan and an electromyography, both of these are diagnostic tests to try to determine possible causes for the symptoms;
- a follow-up appointment with the physician to discuss the results of the diagnostic tests.

Bill and Joan need to do the following things for their mother:

- fill the prescription at a pharmacy;
- make appointments to take their mother to the two treatments;
- make an appointment for the doctor’s follow-up meeting.

For the three appointment times, there are the following preferences and constraints:

- For the two treatments:
 - Bill and Joan would prefer two appointment times that are close together scheduled at one or two nearby places, so that only one person needs to drive, and that person drives only once.
 - Otherwise, they would prefer two appointment times on different days, so that each person needs to drive once.
- The appointment time for doctor’s follow-up check must be later than the appointment times for the two treatments.
- An appointment time must fit the schedule of the person that will drive to the appointment.

Assume that there are the requisite Web Services for finding appointment times and making appointments at the relevant clinics, Bill and Joan could use those services to schedule their mother’s appointments. It would be difficult for Bill and Joan to finish their task with an optimal plan by consulting the Web Services manually, because:

- They may have to try every available pair of close appointment times at any two nearby treatment centers in order to find one that fits their schedules.
- Furthermore, if they first choose an appointment time for one treatment and then find they have to use this same time for the other treatment, then they will have to reschedule the first appointment.

Instead, suppose we use the OWL-S process ontology to encode a description of how to use Web Services to accomplish tasks such as the one faced by Bill and Joan. If we have an automated system which can find an execution path based on these predefined task decompositions, then we can perform Bill and Joan’s Web Services composition task automatically.

3. Background

3.1. OWL-S

In the OWL-S process ontology, operations are modeled as processes. There are three kinds of

```
<owl:Class rdf:ID="PharmacyLocator">
  <rdfs:subClassOf rdf:resource="#process;#AtomicProcess"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="LocationPreference">
  <rdfs:subPropertyOf rdf:resource="#process;#input"/>
  <rdfs:domain rdf:resource="#PharmacyLocator"/>
  <rdfs:range rdf:resource="#concepts;#LocationPreference"/>
</owl:ObjectProperty>
```

processes: *atomic* processes, *composite* processes and *simple* processes. In OWL-S, an *atomic* process is a model of a “single step” (from the point of view of the client) Web Service that is directly executed to accomplish some task. Executing an *atomic* process consists of calling the corresponding Web-accessible program with its input parameters bound to particular values. A *composite* process represents a compound Web Service, i.e., it can be decomposed into other *atomic*, *simple* or *composite* processes. The decomposition of a *composite* process is specified through its control constructs. The set of control constructs includes: **Sequence, Unordered, Choice, If–Then–Else, Iterate, Repeat–Until, Repeat–While, Split** and **Split + Join**. A *simple* process is an abstraction of an atomic or composite process (or of a possibly empty set of these). It is not considered to be directly executable, but provides an abstract view of an action. Like atomic processes, simple processes are, themselves, single-step, but unlike atomic processes, it’s possible to peek at the internal structure of a simple process (if available) or to replace the simple process with an expansion of it.

In the process ontology, each process has several properties, including, (*optional*)*input*, *preconditions*,

(conditional) *outputs* and (conditional) *effects*. *Preconditions* specify things that must be true of the world in order for an agent to execute a service. *Effects* characterize the physical side-effects that execution of a Web-Service has on the world. *Inputs* and *Outputs* correspond to knowledge preconditions and effects. That is, necessary states of our knowledge base before execution and modifications to our knowledge base as a result of the execution. Note that not all services have significant side-effects, in particular, services that are strictly information-providing do not. Here is part of the OWL-S (Version 0.9) definition of an atomic process called PharmacyLocator used in our treatment scheduling example:

The process model of a compound Web Service includes the designation of the top-level composite process corresponding to that service plus a decomposition of that composite process into a structured collection of (perhaps further decomposed) subprocesses.² Web Services composition is sometimes thought of as the process of generating a (potentially) complexly structured composite process description which is subsequently executed. On this model, composite processes are the *output* of composition. In this paper, we take composite processes as *input* to a planner, that is, as descriptions of *how* to compose a sequence of single step actions. Thus, for us, the goal of automated Web Services composition is find a collection of atomic processes instances which form an execution path for some top-level composite process.

3.2. SHOP2

SHOP2 is a domain-independent HTN planning system, which won one of the top four awards out of the 14

² Here, we assume that a compound Web Service always has a complete decomposition bottoming out in atomic processes. Such a composite process is *executable*.

planners that competed in the 2002 International Planning Competition. HTN planning is an AI planning methodology that creates plans by task decomposition. HTN planners differ from classical AI planners in what they plan for, and how they plan for it. The objective 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 operators similar to those of classical planning, and also a set of methods, each of which is a prescription for how to decompose a task into subtasks. Planning proceeds by using methods to decompose tasks recursively into smaller and smaller subtasks, until the planner reaches primitive tasks that can be performed directly using the planning operators.

One difference between SHOP2 and most other HTN planning systems is that SHOP2 plans for tasks in the same order that they will later be executed. Planning for tasks in the order that those tasks will be performed makes it possible to know the current state of the world at each step in the planning process, which makes it possible for SHOP2's precondition-evaluation mechanism to incorporate significant reasoning power and the ability to call external programs. This makes SHOP2 ideal as a basis for integrating planning with external information sources, including Web based ones.

In order to do planning in a given planning domain, SHOP2 needs to be given knowledge about that domain. A SHOP2 knowledge base consists of operators and methods (plus, various non-action related facts and axioms). Each operator is a description of what needs to be done to accomplish some primitive task, and each method tells how to decompose some compound task into a set of partially ordered subtasks.

Definition 1 (Operator). A SHOP2 operator is an expression of the form $(h(v^{\rightarrow}) \text{ Pre Del Add})$ where

- $h(v^{\rightarrow})$ is a primitive task with a list of input parameters v^{\rightarrow} .
- *Pre* represents the operator's preconditions.
- *Del* represents the operator's delete list which includes the list of things that will become false after operator's execution.
- *Add* represents the operator's add list which includes the list of things that will become true after operator's execution.

The expressivity of SHOP2 preconditions and effects are similar to those found in Planning Domain Definition Language (PDDL) [7]. Preconditions contain logical atoms with variables that are either defined in h or existentially quantified. Logical atoms can be combined using the logical connectives such as conjunction, disjunction, negation, implication and universal quantification. *Add* and *Del* lists are generally defined to be a conjunction of logical atoms but conditional expressions and universally quantified expressions can also be used.

Definition 2 (Method). A SHOP2 method is an expression of the form $(h(v^{\rightarrow}) \text{ Pre}_1 T_1 \text{ Pre}_2 T_2 \dots)$ where

- $h(v^{\rightarrow})$ is a compound task with a list of input parameters (v^{\rightarrow}) .
- Each Pre_i is a precondition expression
- Each T_i is a partially ordered set of subtasks.

The meaning of this is analogous to a conditional expression: it tells SHOP2 that if Pre_1 is satisfied then T_1 should be used, otherwise if Pre_2 is satisfied then T_2 should be used, and so forth. A task list consists of task atoms and other task lists. A task list can be defined as *ordered* or *unordered*. The tasks in an *ordered* list must be achieved sequentially whereas tasks in an *unordered* list can be achieved in any order. Nesting of *ordered* and *unordered* task lists can be used to achieve more complex ordering restrictions. A task itself represents an activity to perform and may be either primitive or compound. A primitive task is supposed to be accomplished by a planning operator. A compound task needs to be decomposed into smaller tasks using a method. There may be multiple methods that match a given task so SHOP2 will try each possible decomposition and backtrack if the decomposition ultimately fails.

In addition to the usual logical atoms, preconditions of SHOP2 methods and operators may also contain calls to external programs and assignments to variables. These are useful for integrating planning with queries to information sources on the Web. For example, the following expression

assign $v(\text{call } f \ t_1 t_2 \dots t_n)$

will bind the variable symbol v with the result of calling external procedure f with arguments $t_1 t_2 \dots t_n$.

Definition 3 (Planning problem). A planning problem for SHOP2 is a triple (S, T, D) , where S is initial state, T is a task list, and D is a domain description. By taking (S, T, D) as input, SHOP2 will return a plan $P = (p_1 p_2 \dots p_n)$, that is, a sequence of instantiated operators that will achieve T from S in D .

4. Why HTN planning is suitable for Web Service composition?

There is a clear point where the composition as planning and composition as building up, i.e., “composing,” CompositeProcesses intersect: when the plan itself is a CompositeProcess. This is always trivially the case as a standard SHOP2 plan is a sequence of operators. Furthermore, it is straightforward to extend SHOP2 to generate conditional plans which begin to look like more *interesting* CompositeProcesses. However, the generation of CompositeProcesses by planning is better viewed as the *specialization* of prewritten CompositeProcesses than the authoring of complex, entirely novel programs.

There are several ways in which the HTN approach is promising for service composition:

- HTN encourages modularity. Methods can be written without consideration of how its subtasks will decompose or what compound tasks it decomposes. The method author is encouraged to focus on the particular level of decomposition at hand.
- This modularity fits in well with Web Services. Methods correspond to *recursively composable workflows*. These workflows can come from diverse independent sources and then integrated by the planner to produce situation specific, instantiated workflows.
- Since the planner considers the entire execution path, it has opportunities to minimize various sorts of failures or costs. Most obviously, if the planner finds a plan, one knows that the top level task is achievable with the resources at hand. If the granularity of the services is large enough then it can be considerably easier for a human being to inspect and understand the plan.
- HTN planning scales well to large numbers of methods and operators.

- Some HTN planners (e.g., SHOP2) support complex precondition reasoning, and even the evaluation of arbitrary code at plan time. These features make it straightforward to, integrate existing knowledge bases on the Semantic Web as well as the information supplying Web Services.
- HTN planning provides natural places for human intervention at plan time. The two obvious examples are first, that in preconditions, a code or service call can query a person for special input, and second, if the planner hits a point where it cannot continue de composition, it can request a decomposition of that step from another person, or even a software agent.³

5. From OWL-S to SHOP2

The execution of an atomic process is a call to the corresponding Web accessible program with its input parameters instantiated.⁴ The execution of a composite process ultimately consists in the execution of a collection of specific atomic processes. Instead of directly executing the composite process as a program on an OWL-S interpreter, we can treat the composite process as specification for how to compose a sequence of atomic process executions. In this section, we will show how to encode a composite process composition problem as a SHOP2 planning problem, so SHOP2 can be used with OWL-S Web Services descriptions to automatically generate a composition of Web Services calls.

5.1. Encoding OWL-S process models as SHOP2 domains

In this section, we introduce an algorithm for translating a collection of OWL-S process models K into a SHOP2 domain D . In our translation, we make the following assumption:

Assumption 1. Given a collection of OWL-S process models $K = \{K_1, K_2, \dots, K_n\}$, we assume:

³ For example, the HiCAP [8] system employed SHOP as a component of a mixed initiative system.

⁴ Here, we assume that before the execution of an atomic process the preconditions for executing the atomic process have been satisfied.

- All atomic processes defined in K can either have effects or outputs, but not both. According to the situation calculus based semantics of OWL-S [9], *outputs* characterize knowledge effects of executing Web Services and *effects* characterize physical effects for executing Web Services. An atomic process with only outputs models a strictly information-providing Web Service. And an atomic process with only effects models a world-altering Web Service. In general, we do not want to actually affect the world during planning. However, we do want to gather certain information from information-providing Web Services, which entails executing them at plan time. To enable information gathering from Web Services at planning time, we require that the atomic processes to be either exclusively information-providing or exclusively world-altering.
- There is no composite process in K with OWL-S's **Split** and **Split + Join** control constructs. SHOP2 currently does not handle concurrency. Therefore in our translation, we only consider OWL-S process models that have no composite process using **Split** and **Split + Join** control construct. We also assume only a non-concurrent interpretation of **Unordered** (as permitted by OWL-S). We intend to address how to extend SHOP2 to handle concurrency in the future work.

We encode a collection of OWL-S process definitions K into a SHOP2 domain D as follows:

- For each atomic process with effects in K , we encode it as a SHOP2 operator that simulates the effects of the world-altering Web Service.
- For each atomic process with output in K , we encode it as a SHOP2 operator whose precondition include a call to the information-providing Web Service.
- For each simple or composite process in K , we encode it as one or more SHOP2 methods. These methods will tell how to decompose an HTN task that represents the simple or composite process.

The following algorithm shows how to translate an OWL-S definition of an atomic process with only effects into a SHOP2 operator.⁵

⁵ Conditional effects can be easily encoded into SHOP2 operators. Here, for simplicity, we assume that effects (and outputs) are not conditional.

5.1.1. Translate-atomic-process-effect (Q)

Input: a OWL-S definition Q of an atomic process A with only effects.

Output: a SHOP2 operator O .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for A in Q .
- (2) Pre = conjunct of all preconditions of A , as defined in Q .
- (3) Add = collection of all positive effects of A , as defined in Q .
- (4) Del = collection of all negative effects of A , as defined in Q .
- (5) Return $O = (A(v^{\rightarrow}) Pre Del Add)$.

The above algorithm translates each atomic OWL-S definition into a SHOP2 operator that will simulate the effects of a world-altering Web Service by changing its local state via an operator. Such Web Services will never be executed at planning time, for obvious reasons.

The following algorithm shows how to translate a OWL-S definition of an atomic process with only outputs into a SHOP2 operator.

5.1.2. Translate-atomic-process-output (Q)

Input: a OWL-S definition Q of an atomic process A with only outputs.

Output: a SHOP2 operator O .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for A as in Q .
- (2) Pre = a conjunct of all the preconditions of A , as defined in Q , plus one more pre condition of the form (assign y (call Monitor A if)), where Monitor is a procedure which will handle SHOP2's call to Web Services.
- (3) $Add = y$.
- (4) $Del = \emptyset$.
- (5) Return $O = (A(v^{\rightarrow}) Pre Del Add)$.

The above algorithm translates each atomic OWL-S definition into a SHOP2 operator that will call the information-providing Web Service in its precondition. In this way, the information-providing Web Service is executed during the planning process. The operator for these atomic processes are entirely "book-keeping," thus none of these operators will appear in the final plan.

The following algorithm shows how to translate a OWL-S definition of a simple process into SHOP2 method(s).

5.1.3. Translate-simple-process(Q)

Input: a OWL-S definition Q of a simple process S .

Output: a collection of SHOP2 methods M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for S as in Q .
- (2) Pre = conjunct of all preconditions of S as defined in Q .
- (3) (b_1, \dots, b_m) = the list of atomic and composite processes that realizes or collapse into S as defined in Q .
- (4) for $i = 1, \dots, m$.
 - $M_i = (S(v^{\rightarrow}) Pre b_i)$.
- (5) return $M = \{M_1, \dots, M_m\}$.

The following algorithm shows how to translate a OWL-S definition of a composite process with **Sequence** control construct into a SHOP2 method.

5.1.4. Translate-Sequence-Process(Q)

Input: a OWL-S definition Q of a composite process C with **Sequence** control construct.

Output: a SHOP2 method M .

Procedure:

- (1) (v^{\rightarrow}) = the list of input parameters defined for C as in Q .
- (2) π_{if} = conjunct of all preconditions of C as defined in Q .
- (3) Pre_1 = **Sequence** control construct of C as defined in Q .
- (4) (b_1, \dots, b_m) = the sequence of component processes of B as defined in Q .
- (5) T = ordered task list of (b_1, \dots, b_m) .
- (6) Return $M = (C(v^{\rightarrow}) Pre T)$.

The following algorithm shows how to translate a OWL-S definition of a composite process with **If-Then-Else** control construct into a SHOP2 method.

5.1.5. Translate-If-Then-Else-Process(Q)

Input: a OWL-S definition Q of a composite process C with **If-Then-Else** control construct.

Output: a SHOP2 method M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for C as in Q .
- (2) π_{if} = conditions for **If** as defined in Q .
- (3) Pre_1 = conjunct of all preconditions of C as defined in Q and π_{if} .
- (4) Pre_2 is conjunct of all preconditions of C as defined in Q .
- (5) b_1 = process for **Then** as defined in Q .
- (6) b_2 = process for **Else** as defined in Q .
- (7) Return $M = (C(v^{\rightarrow}) Pre_1 b_1 Pre_2 b_2)$.

The following algorithm translates a OWL-S definition of a composite process with **Repeat-While** control construct into SHOP2 methods.

5.1.6. Translate-Repeat-While-Process(Q)

Input: a OWL-S definition Q of a composite process C with **Repeat-While** control construct.

Output: a collection of SHOP2 methods M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for C as in Q .
- (2) π_{while} = conditions for **While** as defined in Q .
- (3) Pre = conjunct of all preconditions of C as defined in Q .
- (4) b_1 = process for **Repeat** as defined in Q .
- (5) $M_1 = (C(v^{\rightarrow}) Pre C_1(v^{\rightarrow}))$.
- (6) $M_2 = (C_1(v^{\rightarrow}) \pi_{while} (b_1 C_1(v^{\rightarrow}) \emptyset \emptyset))$.
- (7) Return $M = \{M_1, M_2\}$.

Note that M_2 method definition has two condition–task list pairs. The first condition and task list pair ensures that the process inside the loop is iterated as long as the condition is true. When this condition becomes false, SHOP2 will check the second condition which is empty (denoted by \emptyset) thus always true. The task list for this condition is also empty so nothing will be added to the resulting plan. This second pair is just needed to make sure that plan will not fail when the loop condition becomes false.

The following algorithm translates a OWL-S definition of a composite process with **Repeat-Until** control construct into SHOP2 methods.

5.1.7. Translate-Repeat-Until-process(Q)

Input: a OWL-S definition Q of a composite process C with **Repeat-Until** control construct.

Output: a collection of SHOP2 methods M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for C as in Q .
- (2) π_{Until} = conditions for **Until** as defined in Q .
- (3) Pre = conjunct of all preconditions of C as defined in Q .
- (4) b_1 = process for **Repeat** as defined in Q .
- (5) $M_1 = (C(v^{\rightarrow}) \text{ Pre } C_1(v^{\rightarrow}))$.
- (6) $M_2 = (C_1(v^{\rightarrow}) (\text{not}(\pi_{Until})) (b_1 C_1(v^{\rightarrow}) \emptyset \emptyset))$.
- (7) Return $M = \{M_1, M_2\}$.

The following algorithm translates a OWL-S definition of a composite process with **Choice** control construct into a collection of SHOP2 methods.

5.1.8. Translate-Choice-process(Q)

Input: a OWL-S definition Q of a composite process C with **Choice** control construct.

Output: a collection of SHOP2 methods M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for C as in Q .
- (2) Pre = conjunct of all preconditions of C as defined in Q .
- (3) $B = \mathbf{Choice}$ control construct of C as defined in Q .
- (4) (b_1, \dots, b_m) = the bag of component processes of B as defined in Q .
- (5) for $i = 1, \dots, m$.
 - $M_i = (C(v^{\rightarrow}) \text{ Pre } b_i)$.
- (6) return $M = \{M_1, \dots, M_m\}$.

The following algorithm translates a OWL-S definition of a composite process with **Unordered** control construct into a SHOP2 method.

5.1.9. Translate-Unordered-process(Q)

Input: a OWL-S definition Q of a composite process C with **Unordered** control construct.

Output: a SHOP2 method M .

Procedure:

- (1) v^{\rightarrow} = the list of input parameters defined for C as in Q .
- (2) Pre = conjunct of all preconditions of C as defined in Q .
- (3) $B = \mathbf{Unordered}$ control construct of C as defined in Q .

- (4) (b_1, \dots, b_m) = the bag of component processes of B as defined in Q .
- (5) $T = \text{unordered task list of } (b_1, \dots, b_m)$.
- (6) Return $M = (C(v^{\rightarrow}) \text{ Pre } T)$.

The following algorithm translates a collection of OWL-S process models into a SHOP2 domain.

5.1.10. Translate-Process-Model (K)

Input: a collection of OWL-S process models K .

Output: a SHOP2 domain D .

Procedure:

- (1) $D = \emptyset$.
- (2) For each atomic process definition Q in K :
 - If this atomic process has only outputs:
 - $O = \text{TRANSLATE-ATOMIC-PROCESS-OUTPUT}(Q)$.
 - If this atomic process has only effects:
 - $O = \text{TRANSLATE-ATOMIC-PROCESS-EFFECT}(Q)$.
 - Add O to D .
- (3) For each simple process definition Q in K :
 - $M = \text{TRANSLATE-SIMPLE-PROCESS}(Q)$.
 - Add M to D .
- (4) For each composite process definition Q in K :
 - If the process has a **Sequence** control construct
 - $M = \text{TRANSLATE-Sequence-PROCESS}(Q)$.
 - If the process has a **If-Then-Else** control construct
 - $M = \text{TRANSLATE-If-Then-Else-PROCESS}(Q)$.
 - If the process has a **Choice** control construct
 - $M = \text{TRANSLATE-Choice-PROCESS}(Q)$
 - If the process has a **Repeat-While** control construct
 - $M = \text{TRANSLATE-Repeat-While-PROCESS}(Q)$.
 - If the process has a **Repeat-Until** control construct
 - $M = \text{TRANSLATE-Repeat-Until-PROCESS}(Q)$.
 - If the process has a **Unordered** control construct
 - $M = \text{TRANSLATE-Unordered-PROCESS}(Q)$.
 - Add M to D .
- (5) Return D .

To keep the above pseudocode simple, we did not specify the recursive translations within a composite process, e.g., what happens if we have a **Sequence** of **If–Then–Else** or further nestings? Our way for handling this problem is to treat each control construct within a composite process as a composite process. For above example, in our translation, we will have a SHOP2 method for the composite process with **Sequence** control construct and a method for each nested **If–Then–Else** control construct.

Also we did not explicitly describe how our algorithm handles processes with dataflow specification. In OWL-S, a composite process can specify that an output of a composite process is equal to an output of one of its subprocesses whenever the composite process is instantiated. Also, for a composite process with a **Sequence** control construct, one can specify that the output of one subprocess is an input to another subprocesses. SHOP2 does not have the concept of an output and we simply treat outputs as knowledge effects. The output results of a service are recorded in the current state using a special predicate and by assigning a unique number to each instance of a SHOP2 domain's methods and operators. The predicate (*Output Instance Value*) indicates a method or operator instance *Instance* has the value *Value* for the particular output *Output*.

The other aspect of the translation we omitted in the algorithm is the translation of preconditions and effects. The current OWL-S specification (Version 1.0) does not have a concrete syntax for precondition specification. OWL-S 1.1 will support the description of the preconditions and effects of services using OWL statements with variables similar to atoms in the Semantic Web Rule Language (SWRL). These atoms will be combined with logical connectives that are already supported in SHOP2. The translation of such expressions would be syntactically straight-forward but it is also important to preserve the semantics of OWL—a much more challenging task (see Section 7.1).

5.2. Encoding OWL-S Web Services composition problem as SHOP2 planning problem

Narayanan and McIlraith [9] give a formal semantics for OWL-S in terms of the situation calculus [10] and Golog [11]. The situation calculus is a first-order language for reasoning about action and change. In the situation calculus, the state of the world is described by

a - primitive action $\delta_1; \delta_2$ - sequence $cond?$ - test $\delta_1 \delta_2$ - nondeterministic choice of actions δ^* - nondeterministic iteration if $cond$ then δ_1 else δ_2 endif - conditional while $cond$ do δ endwhile - while loop
--

Fig. 1. A subset of Golog constructs to create complex actions that are relevant to OWL-S constructs.

functions and relations (fluents) relativized to a situation s , e.g., $f(x, s)$. The function $do(a, s)$ maps a situation s and an action a into a new situation. A situation is simply a history of the primitive actions performed from an initial, distinguished situation S_0 .

Golog is a high-level logic programming language based on the situation calculus, that enables the representation of complex actions. It builds on top of the situation calculus by providing a set of extralogical constructs (Fig. 1) for assembling primitive actions, defined in the situation calculus, into complex actions that collectively comprise a program, δ . Given a domain theory, D and a Golog program δ , program execution must find a sequence a^{\rightarrow} , such that $D \models Do(\delta, S_0, do(a^{\rightarrow}, S_0))$. $Do(\delta, S_0, do(a^{\rightarrow}, S_0))$ denotes that Golog program δ starting at S_0 will legally terminate in situation $do(a^{\rightarrow}, S_0)$ where $do(a^{\rightarrow}, S_0)$ is used to abbreviate the expression $do(a_n, do(a_{n-1}, \dots, do(a_1, S_0)))$. Thus, a_1, \dots, a_n are the actions that realize Golog program δ , starting in the initial situation, S_0 .

The semantics given in [9] and [12] maps an OWL-S process to a Golog program where atomic processes in OWL-S are mapped to primitive actions in Golog and composite processes in OWL-S are mapped to corresponding complex Golog actions. Using these semantics, we can define the OWL-S service composition problem as follows:

Definition 4 (OWL-S Service composition). Let $K = \{K_1, K_2, \dots, K_m\}$ be a collection of OWLS process models satisfying Assumption 1 (from Section 5.1), C be a possibly composite process defined in K , S_0 be the initial state, and $P = (p_1, p_2, \dots, p_n)$ be a sequence of atomic processes defined in K . Then P is a composition for C with respect to K in S_0 if in action theory, we can prove:

$$\Sigma \models Do(\delta_C, S_0, do(\vec{a}, S_0))$$

where

- Σ is the axiomatization of K and S_0 as defined in action theory;
- δ_C is the complex action defined for C as defined in action theory;
- a_i is the primitive action defined for p_i as defined in action theory;

Note that this definition is for offline planning, i.e., there is no execution of information-providing Web Services during planning. This definition assumes that the initial state contains the complete information for the domain. In reality, this is not the case as we interleave the execution of information-providing services with the simulation of world-altering ones to complete the information in the initial state. Information gathering is done with respect to the initial state so the planning process would yield the same results if all the information-providing Web Services were executed prior to planning. There are some conditions (similar to the IRP assumption [12]) that needs to hold in order to extend this theorem for interleaved execution. We will discuss these conditions at the end of this section.

We will now prove that the plans SHOP2 finds for the OWL-S service composition problem are equivalent to the action sequences found in situation calculus.

We will use the simplified version of SHOP2 algorithm (Fig. 2) during the proof. Since Golog does not provide an *Unordered* construct we will not consider this construct in our proof and in the SHOP2 algorithm we have omitted the details related to unordered tasks. It is possible to define *Unordered* construct in ConGolog (Concurrent Golog) [13] which is an extension to Golog that allows concurrent execution. But since SHOP2 does not allow concurrent processes we cannot use this extension. Also note that in the original Golog formalism complex actions are defined as macro definitions [11] so complex actions do not have preconditions. In our proof, we will show the correspondence to the original Golog approach and assume that in the given OWL-S process model only atomic processes have preconditions.

Theorem 5. Let $K = \{K_1, K_2, \dots, K_m\}$ be a collection of OWL-S process models satisfying Assumption 1 (from Section 5.1), C be a possibly composite process defined in K , S_0 be the initial state, and $P = (p_1, p_2, \dots, p_n)$ be a sequence of atomic processes defined in K . Let $D = \text{TRANSLATE-PROCESS-MODEL}(K)$. Then P is a composition for C with respect to K in S_0 if P is a plan for planning problem (S_0, M_C, D) where M_C is the SHOP translation for process C .

```

1  procedure SHOP2( $s, T, D$ )
2    if  $T$  is empty then return empty plan
3    Let  $t$  be the first task in  $T$ 
4    if  $t$  is a primitive task then
5      Find an operator  $o = (h \text{ Pre } \text{Add } \text{Del})$  in  $D$  such that
         $h$  unifies with  $t$  and  $s$  satisfies  $\text{Pre}$ 
6      if no such  $o$  exists then return failure
7      Let  $s'$  be  $s$  after deleting  $\text{Del}$  and adding  $\text{Add}$ 
8      Let  $T'$  be  $T$  after removing  $t$ 
9      return [ $o$ , SHOP2( $s', T', D$ )]
10   else if  $t$  is a composite task
11     Find a method  $m = (h \text{ Pre}_1 T_1 \text{ Pre}_2 T_2 \dots)$  in  $D$  such that
         $h$  unifies with  $t$ 
12     Find the task list  $T_i$  such that
         $s$  satisfies  $\text{Pre}_i$  and does not satisfy  $\text{Pre}_k, k < i$ 
13     if no such  $T_i$  exists then return failure
14     Let  $T'$  be  $T$  after removing  $t$ 
        and adding all the elements in  $T_i$  at the beginning
15     return SHOP2( $s', T', D$ )
16   end if
17 end SHOP2

```

Fig. 2. A simplified version of the SHOP2 planning procedure.

PROOF. Before giving the proof we should note that there is a representational difference between how SHOP2 and situation calculus describes the state of the world. SHOP2 represents state by a set of ground atoms whereas in the situation calculus, the state of the world is described by relations (fluents) relativized to a situation. For example, $f(x^{\rightarrow})$ is true at some point in the planning process when that atom occurs in SHOP2's "state" (e.g., the set of ground atoms). In the situation calculus, truth value for that relation is relative to a specific situation argument, e.g., $f(x^{\rightarrow}, s)$. The changes to the state in SHOP2 is done by adding or deleting atoms from the state whereas situation calculus defines successor state axioms to define the truth values for the fluents in different situations. Apart from this representational difference, there is an equivalence between SHOP2 state and situations, e.g., $f(x^{\rightarrow})$ is true in the initial state of SHOP2 if $f(x^{\rightarrow}, S_0)$ is true in situation calculus. Applying the effects of an operator will also preserve this equivalence. It is easy to verify that the truth value for the predicate $f(x^{\rightarrow})$ after applying the effects of an operator will be equal to the truth value of $f(x^{\rightarrow}, do(a, s))$ when a is the corresponding situation calculus action and the starting states are equivalent. In general, when the same sequence of actions/operators are applied to a situation/state, the state of the world in the final situation/state will be the same. Throughout the proof, we will use this equivalence and use the same name to denote world states in both notations when the meaning is clear. The proof of the theorem is by induction. \square

Hypothesis. For a given OWL-S process C , P is a plan for the planning problem (S_0, M_C, D) if $\Sigma| = Do(\delta_C, S_0, do(a^{\rightarrow}, S_0))$ where $a^{\rightarrow} = [a_1, a_2, \dots]$ is the sequence of primitive actions in situation calculus that corresponds to the sequence of SHOP2 operators in P .

Base Case: Suppose A is an atomic OWL-S process and a is the corresponding primitive action in situation calculus and o_A is the corresponding SHOP2 operator. Then in Golog it is defined that

$$Do(a, s, s') = Poss(a, s) \wedge s' = do(a, s)$$

It means when the preconditions for the process is satisfied with respect to situation s then the primitive action sequence we will get for this simple program

will have only one element, namely $a^{\rightarrow} = [a]$. As seen in line 9 of SHOP2 algorithm, the plan for a primitive task will return the plan that includes the operator instance when the preconditions of that operator are satisfied (the recursive call will return empty list as there are no more tasks in the list). Thus, the plan returned by SHOP2 is $[o_A]$ which is equivalent to the situation calculus result.

Inductive Step: We will do a case by case analysis for each of the control constructs in the process model to show that our translation and resulting plans SHOP2 finds are correct.

Choice: Suppose C is a composite OWL-S process defined as a *Choice* of two⁶ other processes C_1 and C_2 . The SHOP2 translation for C will yield two methods $M_1 = (C \emptyset M_{C_1})$ and $M_2 = (C \emptyset M_{C_2})$. Note that the SHOP2 methods have no preconditions (\emptyset is used for preconditions) because we have assumed that composite processes cannot have preconditions. Corresponding Golog program for C is $\delta_C = \delta_{C_1} | \delta_{C_2}$ and the semantics is defined as

$$Do(\delta_{C_1} | \delta_{C_2}, s, s') = Do(\delta_{C_1}, s, s') \vee Do(\delta_{C_2}, s, s')$$

The disjunction means any a^{\rightarrow} that is a valid action sequence for either δ_{C_1} or δ_{C_2} will also be a valid sequence for δ_C . From our hypothesis we know for each action sequence a^{\rightarrow} that satisfies δ_{C_1} (or δ_{C_2}) we have a valid SHOP2 plan P_{C_1} (or P_{C_2}). The nondeterministic choice in SHOP2 algorithm (line 11) shows that when a plan is being sought for C , the solution for any matching method instance, in this case M_1 and M_2 , will be returned as a result. This ensures that when SHOP2 is asked to find all the plans for C , both P_{C_1} and P_{C_2} will be returned proving the equivalence to the answer in situation calculus.

Sequence: Suppose C is a composite OWL-S process defined as a *Sequence* of two other processes C_1 and C_2 . The SHOP2 translation for C will yield one method $M_C = (C \emptyset (M_{C_1} M_{C_2}))$. Corresponding Golog program for C is $\delta_C = \delta_{C_1}; \delta_{C_2}$ and the semantics is defined as

$$Do(\delta_{C_1}; \delta_{C_2}, s, s') = (\exists s^*)(Do(\delta_{C_1}, s, s^*) \wedge Do(\delta_{C_2}, s^*, s'))$$

Suppose that situation s^* represents a history of the action sequence a_1^{\rightarrow} . If the action sequence

⁶ Golog choice operator $|$ is defined for two operands. A choice of more operands could be done by nested $|$ operators which would not effect our proof here.

recorded between situations s^* and s' is $a_2 \rightarrow$ then the final situation s' represents the concatenated sequence $a \rightarrow = [a_1 \rightarrow, a_2 \rightarrow]$. Calling SHOP2(s, M_{C_1}, D)

translation to be $M_C = (C \text{ cond } (C_1 \ C) \ \emptyset \ \emptyset)$. Corresponding Golog program for C is $\delta_C = (\mathbf{while \ cond \ do} \ \delta_{C_1} \ \mathbf{endWhile})$ and the semantics is defined as

$$Do(\mathbf{while \ cond \ do} \ \delta_{C_1} \ \mathbf{endWhile}, s, s') = Do([[(\text{cond?}; \delta_{C_1})]^*; \neg \text{cond?}], s, s')$$

will return P_{C_1} and from our hypothesis we know that it is equivalent to the action sequence $a_1 \rightarrow$. We also know that calling SHOP2(s^*, M_{C_2}, D) will return a plan P_{C_2} that is equivalent to the action sequence $a_1 \rightarrow$. SHOP2 algorithm shows that (line 14) when a task (in this case M_C) is removed from the input task network T , it is replaced with its sub-elements (in this case M_{C_1} and M_{C_2}). The tasks to solve are selected from T in the order they were added (line 3) so the resulting plan for SHOP2(s, M_C, D) will actually be the concatenation of P_{C_1} and P_{C_2} which is equivalent to the sequence $a_1 \rightarrow$

If–Then–Else: Suppose C is a composite OWL-S process defined with a *If–Then–Else* control construct and cond is the condition for the if statement, C_1 is the process in the then part and C_2 is the process in the else part. The SHOP2 translation for C will yield one method $M_C = (C \text{ cond } M_{C_1} \ \emptyset \ M_{C_2})$. Corresponding Golog program for C is $\delta_C = (\mathbf{if \ cond \ then} \ \delta_{C_1} \ \mathbf{else} \ \delta_{C_2} \ \mathbf{endif})$ and the semantics is defined as

$$\begin{aligned} & Do(\mathbf{if \ cond \ then} \ \delta_{C_1} \ \mathbf{else} \ \delta_{C_2} \ \mathbf{endif}, s, s') \\ &= Do((\text{cond?}; \delta_{C_1}), s, s') \vee Do((\neg \text{cond?}; \delta_{C_2}), s, s') \\ &= (\text{cond}[s] \wedge Do(\delta_{C_1}, s, s')) \vee (\neg \text{cond}[s] \wedge Do(\delta_{C_2}, s, s')) \end{aligned}$$

The expression $\text{cond}[s]$ evaluates to true whenever the fluent cond is true in situation s . Suppose $a_1 \rightarrow$ is the action sequence for the situation δ_{C_1} and $a_2 \rightarrow$ is the action sequence for the situation δ_{C_2} . If s satisfies cond then the result for δ_C will be $a_1 \rightarrow$ otherwise result will be $a_2 \rightarrow$. From our hypothesis we know for any possible $a_1 \rightarrow$ (or $a_2 \rightarrow$) we have a valid SHOP2 plan P_{C_1} (or P_{C_2}). When we call SHOP2(s, M_C, D), the algorithm will check the conditions in the method definition (line 12), cond and \emptyset in this translation. If cond is satisfied algorithm returns P_{C_1} and otherwise returns P_{C_2} which is equivalent to the result in situation calculus.

Repeat–While: Suppose C is a composite OWL-S process defined with a *Repeat–While* control construct and cond is the condition for the while statement and C_1 is the process in the loop body. As we have assumed that composite processes do not have preconditions, without losing generality, we can simplify the SHOP2

This definition includes the nondeterministic iteration operation $*$ which has a second-order semantics [11]. We will use the restricted version of Golog as defined in [12] where the iterations has a limit k . This restriction eliminates the problems caused by unlimited looping and enables us to define a first-order semantics.

Assume the iteration runs k times. When $k = 0$, the above formula will simplify to $Do(\neg \text{cond?}, s, s')$ which returns an empty action sequence in situation calculus. This new formula also implies condition cond is false in the initial situation s . When SHOP2 is trying to solve M_C , since cond is false the algorithm will choose (line 12) the second condition–task list pair (note that the second condition in M_C is \emptyset which is always true). The second task list is \emptyset so SHOP2 will return an empty plan as well. Suppose $a \rightarrow$ is a valid action sequence for δ_{C_1} . From our hypothesis we know for each action sequence $a \rightarrow$ that satisfies δ_{C_1} we have a valid SHOP2 plan P_{C_1} . In the general case, when $k > 0$, the Golog formula becomes $Do([\text{cond?}; (\delta_{C_1})^1; \dots; \text{cond?}; (\delta_{C_1})^k; \neg \text{cond?}], s, s')$ hence the action sequence will be $[a_1 \rightarrow, \dots, a_k \rightarrow]$. Note that action sequence for each step of iteration may be different, for example when δ_{C_1} contains nondeterministic choices. We also know that cond will be true in situations s, s_1, \dots, s_{k-1} and false in situation S_k . When SHOP2 is searching a plan for M_C , the first condition (cond) will evaluate to true and SHOP2 will chose the first task list ($C_1 \ C$). Solving the first task C_1 will add P_1 to the plan and solving second task C will recursively continue until cond fails. Since, initial states are equal and plan prefixes are same, cond will not hold after k th iteration. At this point, algorithm will chose the second condition–task list pair (empty task list) which will conclude the recursion and the plan returned will be $[P_1, \dots, P_k]$. At each step of the iteration we will have the equivalent world states so the action sequence a_i and plan P_i will be equivalent due to our hypothesis. Therefore, the final plan and the final action sequence will be equivalent.

Repeat-Until: The proof for this case will be very similar to the above proof for *Repeat-While* construct.

Our proof did not include the effects of executing information-providing services during planning. Information gathering during planning is equivalent to the Middle Ground execution (MG) for sensing actions in the Golog approach [12]. In both cases, planning starts with an incomplete initial state and executing sensing actions adds new knowledge to the state. As long as the information retrieved from the services does not change over the course of planning, we would still have the equivalence of world states in both representations and it would be straight-forward to extend the proof for this case.

The correctness of MG depends on the Invocation and Reasonable Persistence (IRP) assumption [12]. Intuitively, IRP assumption says that (1) information-providing services should be executable in the initial state, and (2) information gathered from these services cannot be changed by external or subsequent actions. The first condition follows from the fact that information gathering is done with respect to the initial state. The second condition assumes no external source will change the gathered information during the planning process but also prohibits the planner from changing the gathered information as well. This is to prevent problems such as this one: in our example domain (see Section 2), a Web Service is executed to get the available appointment times from a hospital. Then planner simulates scheduling an appointment at one of the available time slots. If the information-providing service is executed again and the available appointment times (which have not yet been changed) are added to the knowledge base then there would be a problem because planner would be able to schedule another appointment in the same time slot. The IRP prohibits the second step (changing the information retrieved) to overcome this problem. This solution is certainly very restrictive and obviously our example domain violates this assumption. For this reason, our solution is to prohibit the last step where the same information-providing service is executed more than once.

To establish the soundness and completeness of our approach we have the following assumptions about the information-providing Web Services:

- executable (in the initial state with all parameters grounded);
- terminable (with finite computation);
- repeatable (with same result for the same call during the planning process).

We also assume that the information that is returned from different Web Services are disjoint, i.e., no two services return the same information. These assumptions guarantee that gathered information can only be changed by the actions planner simulates. Also there is no way that this simulated change will be undone by another information gathering step as long as we execute each information-providing Web Service at most once. Note that we do not need to run the same service twice since the information is guaranteed to be same each time due to repeatability assumption.

One other thing to note is that, different from the Golog approach, we do not allow the information-providing services to appear in the final plan since our translation methodology maps them to “book-keeping” operators. However, this is just a style difference as in the Golog approach a post-processing step is suggested to find the world-altering services for the execution of the resulting plan. In some situations, it could still be valuable to include the information-providing services in the plan so a prudent action could verify if the information-providing services still return same information. This could be easily achieved in our system by changing the *TRANSLATE-ATOMIC-PROCESS-OUTPUT* procedure to generate a standard operator rather than a “book-keeping” operator (translation rules for precondition and effect).

6. Implementation

To realize our ideas, we started with an implementation of a OWL-S to SHOP2 translator. This translator is a Java program that reads in a collection of OWL-S process definitions and outputs a SHOP2 domain. As shown in the translation algorithm in Section 5.1, when planning for any problem in this domain, SHOP2 will actually call the information-providing Web Services to collect information while maintaining the ability of backtrack by merely simulating the effect of world-altering Web Services. The output of SHOP2 is a sequence of world-altering Web Services calls that can be subsequently executed.

We built a monitor which handles SHOP2's calls to external information-providing Web Services during planning (Fig. 3). We wrote a OWL-S Web Services executor which communicates with SOAP based Web Services described by OWL-S groundings to WSDL descriptions of those Web Services. Upon SHOP2's request, the monitor will call this OWL-S Web Services executor to execute the corresponding Web Service. Since the information-providing services are always defined as atomic processes, the service is executed by invoking the WSDL service in the grounding. The monitor also caches the responses of the information-providing Web Services to avoid invoking a Web Service with same parameters more than once during planning. This will save the network communication times and improve planning efficiency, and establishes the repeatability condition required for proving SHOP2's soundness and completeness. Also information can only be added into the current state if it has not been changed by the planner. We assume that the cached information will not be changed by other agents during planning and we will generalize this in our future work.

We also built a SHOP2 to OWL-S plan converter, which will convert the plan produced by SHOP2 to OWL-S format which can be directly executed by the OWL-S executor.

We ran our scenario from Section 2 on this system. In doing so:

- Our system communicated with real Web Services. Unfortunately, the current Web Services available on the Web have only WSDL descriptions without any semantic mark-up. Therefore, we created OWL-S mark-up for the WSDL descriptions of these online services. For some services it was necessary to create even the WSDL description, e.g., for CVS Online Pharmacy Store. It was not possible to use real services for some of the services either because none was available as Web Services, e.g., a doctor's agent providing the patient's prescription, or it was infeasible to use a real Web Service for the demo, e.g., making an appointment with a doctor each time the program is executed. For these services, we implemented Web Services to simulate these functionalities.
- We built Web Services that allow the access to user's personal information sources. For example, it is necessary to learn the user's schedule to be able to generate a plan for the example task in our demo. It is possible to get this information from the sources available on the user's machine such as a Personal Information Manager like Microsoft's Outlook. We

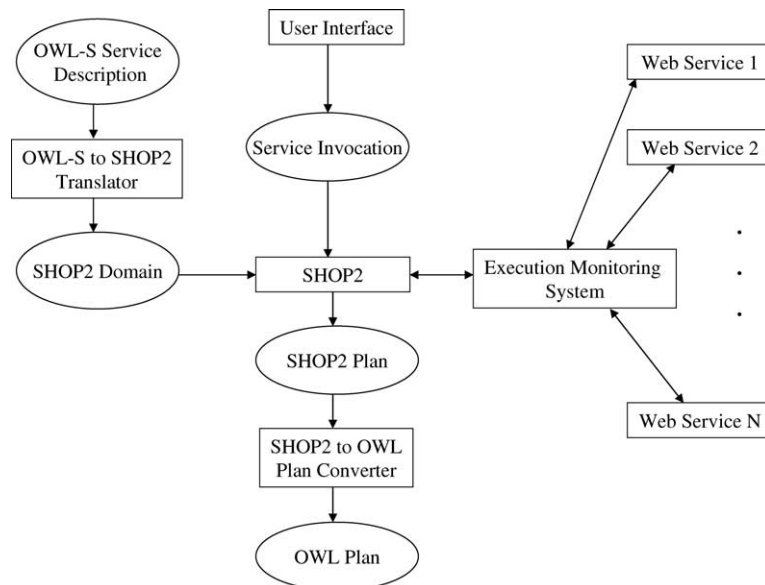


Fig. 3. System Architecture.

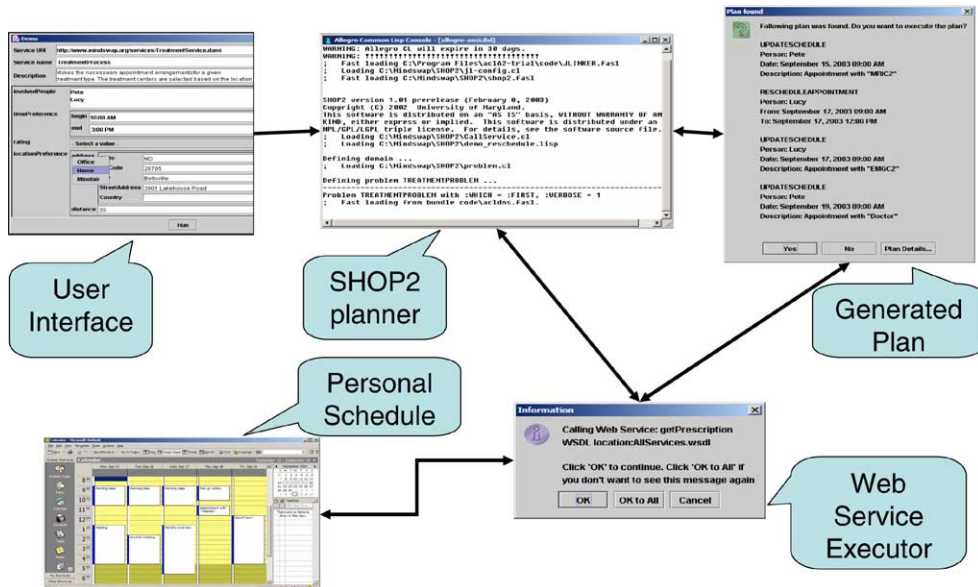


Fig. 4. Example running result.

have implemented “local” SOAP based services that will retrieve this kind of information. WSDL and OWL-S descriptions are also generated for these local services so that they can be composed and executed in the same way as other remotely available services.

Finally, some information gathering services were implemented as direct Java calls from SHOP2 over a Java/SHOP2 bridge. For example, we have a service which asks the user for acceptable distances to the treatment center by popping up a window on the user’s client to accept input. Changing the data entered at this point will possibly yield a different plan to be generated allowing the planner produce custom plans depending on personal preferences.

- We also encoded a description of how to compose Web Services for tasks such as the one faced by Bill and Joan in section 2 in OWL-S. The description is given as a OWL-S composite process that is composed of several other composite processes that are defined as sequence, choice or unordered processes. This OWL-S description constitutes the top level composite process described in Section 5.1 and is translated into a SHOP2 domain for planning. We en-

code most of the constraints mentioned in Section 2 as Web Service preconditions. Right now, there is no standard process modeling language for specifying Web Service preconditions. Therefore, we directly encode the Web Services preconditions in SHOP2 format.

Fig. 4 shows the various components of the system⁷ and the results achieved from a sample run of the example domain. The user starts with a simple user interface where an OWL-S service description for any desired task can be loaded. When the service description for the example domain is selected, a form to enter the required parameters for the task is presented to the user. This form is generated based on the ontologies used to describe the input parameters of the service. The UI will also automatically fill out some of the fields such as the home address from a user specified knowledge base.

Once all the input parameters are provided SHOP2 starts the planning process using the domain description obtained from the translation of the OWL-S files. Note that the service selected in the UI is specified by

⁷ This system was demonstrated in the Developer’s Day of the 12th WWW conference in Budapest, Hungary.

an “abstract” task list, that is, a set of tasks which can be achieved in a variety of ways. In order to “execute” (it would be better to say, “perform”, or “achieve”) this service we must decompose these abstract tasks into actions (services) that we can actually invoke. SHOP2 decomposes the top level task into smaller subtasks, and of course there may be multiple different decompositions for any given task. For example, one decomposition for the top level task yields a task to schedule two appointments on the same day for the same person whereas another decomposition will yield a task to schedule two appointments on two different days for two different drivers (see [Section 2](#) for more information on domain characteristics). Another example abstract task is to find the availability of the prescribed medicine in an online pharmacy store. A decomposition for this task will include all the different Web Services for different online stores. These decompositions are statically given in the OWL-S service descriptions but one can imagine a more dynamic setting where a Web Service repository is queried for possible decompositions.

The SHOP2 planner will execute the information-providing Web Services to gather the necessary information for plan generation, e.g., get the available appointment times from hospitals. Based on the collected information the planner will, if possible, produce a plan that is a valid decomposition of the top level task. This plan is simply a sequence of atomic, directly executable Web Services such as “order the medicine from the online pharmacy store,” “make the appointment in the hospital for the treatment,” and “update my personal calendar with the appointment info.” User has the option to view the details of the plan, reject the plan if desired, and re-plan with a new set of constraints.

To test the effectiveness of our approach, we have run SHOP2 on several instances of the example problem. These problem instances varied from cases where it was easy to schedule satisfactory appointments to a case in which no nearby treatment centers had treatment time slots that were close together, so that Bill and Joan would both have to drive Mom for treatments on separate days. In all of these cases, SHOP2 was easily able to find the best possible solution. [Fig. 4](#) shows a snapshot of the running system and the interaction between different components of the system.

7. Discussion

7.1. Using Semantic Web knowledge bases

SHOP2 represents the state of the world as a set of ground logical atoms. SHOP2 uses axioms which are generalized versions of Horn clauses to infer conditions that are not explicitly asserted in the current state. SHOP2’s theorem prover makes a closed-world assumption. In contrast, Semantic Web, and particularly OWL, has an open-world assumption. The inferences in OWL are done with respect to the OWL Semantics [14]. OWL DL species of the language can be directly mapped to SHION (D) Description Logic (DL) [15] which itself is a decidable subset of first-order logic.

Unfortunately, it is not possible to directly express the semantics of OWL DL using SHOP2 axioms. Therefore using SHOP2’s theorem prover directly causes us to lose the inferencing capability normally an OWL reasoner possesses. Furthermore, the size of the data involved in the planning process over Semantic Web will be much bigger than the ones encountered in classical planning problems. The state of the world consists of all the information coming from ontologies either stored locally or found remotely on the Web. Therefore, the theorem prover should be able to work in Web scale, with thousands or maybe hundreds of thousands of facts and axioms. SHOP2’s inferencing capabilities are not satisfactory for the expressivity and the scalability needed to deal with knowledge bases found on Semantic Web.

It is possible to completely replace the theorem prover SHOP2 uses with a new one. As long as the theorem proving is decidable and the theorem prover is sound and complete then the soundness and completeness of the planner is ensured. So, we can use an OWL reasoner to reason about the state of the world. On this model, the state will be simply represented as an OWL knowledge base. The precondition checking is equivalent to querying the knowledge base and applying effects is equivalent to adding and deleting facts from the knowledge base. If we restrict ourselves to the OWL DL language then we can use sound and complete DL reasoners for this purpose. Also there exists DL reasoners specialized to handle very large knowledge bases [16]. Therefore, we can solve both the expressivity and scalability problems.

Using a DL reasoner inside SHOP2 planner brings out some issues that classical planning theory has not addressed thoroughly. In general, classical planners do not let axiomatic inference at all or only allow a restricted form of inference. For example, secondary relations, relations whose values can be deduced from other relations, are allowed to appear only in preconditions but not in effects to avoid certain complications [p. 42, 17]. Any property in OWL that is defined to have an inverse property can be seen as a secondary relation because the value for that property can be deduced from its inverse property. Either the planner should not accept operator descriptions that use these properties in effects or it should define the semantics associated with it. The semantics may require that if a relation is in the delete list and the property used in the relation has an inverse property then the inverse relation will also be deleted.

As an initial attempt to investigate the applicability of the idea, we have incorporated the OWL DL reasoner Pellet [18] into SHOP2. To avoid the problems mentioned above we have considered only a restricted set of process definitions where preconditions and effects consist of ABox expressions and effects do not mention secondary relations. It was possible to represent the process descriptions used in the example defined in Section 2 with these restrictions. Our initial observations showed that using a DL reasoner increased the amount of time required for planning but overall planning time was still dominated by the time spent for remote Web Service execution. Of course, the reasoning time is related to the structures of the ontologies used and having very complex definitions could effect the reasoner performance significantly. As a future work we are continuing to investigate this in detail along with the effects of allowing more expressive DL constructs in operator definitions.

7.2. Information gathering during planning

There is a fundamental difference between exclusively information-providing and possibly world-altering atomic processes. We typically want to execute information-providing atomic processes at various points in the planning process, while we never want to execute world-altering ones. Contrariwise, at composition execution time, the primary interest is in the execution of world-altering processes. Indeed, in our sys-

tem we do not include any information-providing processes in compositions. Furthermore, currently we do not permit world-altering processes to be information-providing, at least in the sense that they must have no outputs. This simplification made the system fairly easy to implement without substantial modification of SHOP2.

However, mapping information-gathering processes to so-called “book-keeping” operators is somewhat un-aesthetic. In the translation algorithm we described, for each atomic process that does not have any effects a book-keeping operator is created with a precondition that contains the external call to execute the service and an effect to assert the output results as knowledge effects. The book-keeping operator appears as a subtask in the method definition that uses the result of that service. But, these operators are treated specially by SHOP2 and they never appear in the resulting plans.

It is also possible to directly encode the information-providing operators in the preconditions of the calling methods. The external call to service execution would be put into the method’s precondition instead of the intermediate book-keeping operator. The output of the information-providing service would be stored in a local variable using SHOP2’s *assign* feature. We don’t need to store results globally since by definition only the enclosing process should be able to access the results of a subprocess. Using local variables proves to be a more efficient way to handle outputs since the overall number of facts stored in the current state are not effected by the information gathering services.

This different encoding technique has some consequences. For example, normally it is possible to define preconditions for information-providing services. While the book-keeping operators can be used to represent these conditions, the new encoding method does not keep this information. As far as the correctness of the plan is concerned, this is not really a problem. We can directly execute information-providing services and if the precondition of that service is not satisfied the service will simply fail to execute. Checking preconditions is more efficient than trying to execute the service. For typical public information-providing services, there are no adverse consequences to trying to execute the services. In a situation where attempting a information Web Service call was expensive or harmful, we would have to fall back on our prior encoding.

Another issue related to the performance arises when information gathering and world altering services are used together inside sequences. For example, an information gathering service may be defined in between two world altering actions. When this information providing service is encoded in the precondition of the method it will be evaluated before both of the world altering services. This will not effect the resulting plan in any way but may have some impact on the performance. If during planning process it turns out that the first world altering action is not applicable in the current state then the time spent to execute the information gathering service is wasted.

So far we have only considered the cases where we explicitly know which services will provide the information needed for the given task. But actually information gathering itself can be seen as another planning problem [19]. As discussed in the previous section, precondition checking is reduced to query answering on Semantic Web. If the information required to answer the query is not present then we can search for the services who can supply the necessary data. This process can be done as a goal oriented planning process [19] and SHOP2 could call another planner for this purpose. It is also possible that information providing services have a hierarchical structure similar to the world altering ones. Then we can use SHOP2 recursively to first generate a plan for information gathering step, execute this plan to get the information and then use this information to solve the initial planning problem.

8. Related work

McIlraith and coworkers [9,12] proposed an approach to building agent technology based on the notion of generic procedures and customizing user constraints. They adapt and extend the Golog language to enable programs that are generic, customizable and usable in the context of the Web. They augment a ConGolog interpreter that combines online execution of information-providing services with offline simulation of world altering services. Our approach is very similar to this. A general logic-based system has the ability to do arbitrary reasoning about the theory but in general we suspect that a logic based approach will not be as efficient as an HTN planner.

Matskin and Rao [20] applies software synthesis and composition methods to Web Services composition. Their work is based on similarities between Web Service composition and component-based system development in software engineering. They use OWL-S for service descriptions and adopt Structural Synthesis Program (SSP) method for automated service composition. Service composition is based on the input–output information of services components and requires little domain knowledge. This approach treats each service as an atomic entity without inspecting the internal process model and therefore lacks the ability to reason about different decompositions in a composite process.

RETSINA [21] is an open multi-agent system that provides infrastructure for different types of deliberative, goal directed agents. RETSINA system includes a planner based on the HTN planning paradigm. The RETSINA planner also extends HTN planning by adding interleaving of planning and execution which basically allows the actions execute before a plan is completely formed, similar to our approach. Paolucci et al. [22] describes using RETSINA planner in the context of creating autonomous Web Services that can automatically interact with each other. However, authors do not give details about how HTN planning is employed in the system. It is not clear whether OWL-S Process Model was used or planning domain was given a priori to the planner agent. For this reason, we cannot make a comparison with our approach.

9. Conclusion

In this paper, we have defined a translation from OWL-S process models to the SHOP2 domains, and from OWL-S composition tasks to SHOP2 planning problems. We have described our implemented system which performs this translation, uses an extended SHOP2 implementation to plan with and over the translated domain, and then executes the resulting plans. In the process of defining the translation and building the system, we observed that:

- In our current approach, the planner always executes output producing actions as it plans. While this is fine for many situations, it may not always be appropriate. For example, the execution of some Web Services may take a very long time. It would be better

if the planner could continue to plan while waiting for this information.

- In our paper, we assume that all effects are physical. In complex situations, there may be other changes, such as in the mental states of the agents involved, that are not modeled. We will explore this problem in our future work.
- Information providing (whether exclusively so, or not) is likely to be a significant fraction of the available and salient Web Services. Many Web contexts seem to be *information rich* but *action poor*. In such environments, we would want to reconsider the strict partition of services into exclusively information providing and output free. For example, world-altering services with outputs might supply information needed to decide subsequent courses of action. Clearly, such a service should not be executed at planning time, which suggests that we will need to investigate generating conditional plans by SHOP2 style HTN planning.

Conditional plans will also help mitigate the constraint on information change during planning. Currently, both for theoretical and practical reasons, we only execute an information providing process once during planning for any given input, and subsequently retrieve a cached result. Given SHOP's speed, this is not that unreasonable a restriction for many cases, but conditional plans would permit planning for various contingencies.

These considerations raise a host of issues regarding plan time versus composition execution time execution of information providing processes, including those of deciding which such processes to execute only during planning, only during plan execution, and during both. Furthermore, in complex, long running planning sessions, it might make sense to refresh the monitors cache for certain services at intervals. Presumably, OWL-S descriptions will be enriched to help support the requisite analysis. We intend to explore these issues in future work.

- Compared the complexities raised above, composite processes raise no additional or special problems—encoding them as SHOP2 methods seems correct and straightforward, modulo the need to extend SHOP2 to handle concurrency.
- Simple processes are the odd duck of the lot. Though various members of the OWL-S coalition have sug-

gested, in conversation, that simple processes were intended to support HTN planning, we found them neither necessary, nor convenient, nor useful. In part, their lack of a clear semantics, particularly with regard to the relationship of their inputs, outputs, preconditions, and effects to those of their corresponding atomic or composite processes. Furthermore, while the language of the technical overview [3] suggests that a given simple process can be a view of one atomic process or one composite process, but not both, neither the language nor the ontology actually require this restriction. We speculated that this would make simple processes useful for specifying a range of alternative composition paths, but it was not clear that this was really more convenient (for our purposes) than using the **Choice** control construct.

Acknowledgments

The authors would like to thank Sheila McIlraith for her insightful comments about the paper. This work was supported in part by Air Force Research Laboratory grant F30602-00-2-0505. This work was also supported in part by the Army Research Laboratory, the Defense Advanced Research Projects Agency, Fujitsu Laboratory of America College Park, Lockheed Martin Advanced Technology Laboratories, the National Science Foundation, the National Institute of Standards and Technology, and NTT Corp.

References

- [1] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, L.A. Stein, Web Ontology Language (OWL) Reference, W3C Recommendation 10, February 2004, <http://www.w3.org/TR/owl-ref>.
- [2] I. Horrocks, F. van Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, P. Hayes, J. Hefflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein, DAML+OIL, 2001, <http://www.daml.org/2001/03/daml+oil-index.html>.
- [3] OWL Services Coalition, OWL-S: Semantic Markup for Web Services, OWL-S White Paper, 2003, <http://www.daml.org/services/owl-s/0.9/owl-s.pdf>.
- [4] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, S. Mitchell, Total-order planning with partially ordered subtasks, in: Proceedings

- of the Seventeenth International Joint Conference on Artificial Intelligence, Seattle, 2001.
- [5] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, F. Yaman, SHOP2: an HTN planning system, *J. Artif. Intell. Res.* 20 (2003) 379–404.
- [6] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, *Scientific Am.* 284 (5) (2001) 34–43.
- [7] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—The Planning Domain Definition Language, 1998, URL citeseer.ist.psu.edu/ghallab98pddl.html.
- [8] D.W. Aha, L.A. Breslow, H. Munoz-Avila, D.S. Nau, R. Weber, HICAP: hierarchical interactive case-based architecture for planning, 1999.
- [9] S. Narayanan, S. McIlraith, Simulation verification and automated composition of web services, in: *Proceedings of the Eleventh International World Wide Web Conference, Honolulu, Hawaii, 2002*.
- [10] R. Reiter, *Knowledge, Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, The MIT Press, 2001.
- [11] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, R.B. Scherl, GOLOG: a logic programming language for dynamic domains, *J. Logic Programm.* 31 (1-3) (1997) 59–83.
- [12] S. McIlraith, T. Son, Adapting Golog for composition of semantic web services, *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning, Toulouse, France, 2002*.
- [13] G.D. Giacomo, Y. Lesperance, H.J. Levesque, Congolog, a concurrent programming language based on the situation calculus, *Artif. Intell.* 121 (1–2) (2000) 109–169.
- [14] P.F. Patel-Schneider, P. Hayes, I. Horrocks, OWL Web Ontology Language Semantics and Abstract Syntax, W3C Proposed Recommendation 15 December, 2003, <http://www.w3.org/TR/owl-semantics/>.
- [15] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (Eds.), *The Description Logics Handbook: Theory, Implementations, and Applications*, Cambridge University Press, Cambridge, 2003.
- [16] V. Haarslev, R. Moller, High performance reasoning with very large knowledge bases: a practical case study, *IJCAI (2001)* 161–168.
- [17] M. Ghallab, D. Nau, P. Traverso, *Automated Planning: Theory and Practice*, Morgan Kaufmann, 2004.
- [18] Pellet, Pellet – OWL DL Reasoner, 2003, <http://www.mindswap.org/2003/pellet>.
- [19] C.A. Knoblock, Planning, executing, sensing, and replanning for information gathering, *IJCAI (1995)* 1686–1693.
- [20] M. Matskin, J. Rao, Value-added web services composition using automatic program synthesis, in: *Web Services, E-Business, and the Semantic Web, CAiSE, International Workshop, Canada, 2002*.
- [21] K. Sycara, J.A. Giampapa, B.K. Langley, M. Paolucci, The retsina mas, a case study, in: A. Garcia, C. Lucena, F. Zambonelli, A. Omici, J. Castro (Eds.), *Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications*, vol. LNCS 2603, Springer-Verlag, 2003, pp. 232–250.
- [22] M. Paolucci, K. Sycara, T. Kawamura., Delivering semantic web services, in: *Proceedings of the Twelfth International World Wide Web Conference, Budapest Hungary, 2003*.