Planning for Semantic Web Services

Evren Sirin[1] and Bijan Parsia[2]

[1] University of Maryland,
Computer Science Department,
College Park MD 20742, USA
evren@cs.umd.edu
[2] University of Maryland, MIND Lab, 8400 Baltimore Ave,
College Park MD 20742, USA
bparsia@isr.umd.edu

Abstract. Using Semantic Web ontologies to describe Web Services has proven to be useful for various different tasks including service discovery and composition. AI planning techniques have been employed to automate the composition of Web Services described this way. Planners use the description of the preconditions and effects of a service to do various sorts of reasoning about how to combine services into a plan. OWL-S 1.1 will support the description of the preconditions and effects of services using OWL statements similar to atoms in Semantic Web Rule Language (SWRL). Thus, planners are required to understand the semantics of OWL in order to evaluate such preconditions. However, planners typically support only fairly limited reasoning capabilities which cannot handle the expressivity of Semantic Web ontologies. In particular, planners typically make the closed world assumption, whereas OWL has open world semantics. In this paper, we demonstrate how an OWL reasoner can be integrated with an AI planner to overcome these problems. We identify the challenges of writing the service descriptions and reasoning about them when OWL is used to describe preconditions and effects. We also investigate the efficiency of such an integrated system and show how OWL reasoning can be optimized for this system. Finally, we present the performance results of our prototype implementation.

1 Introduction

The Semantic Web vision is of a world where loosely coupled, independently evolving ontologies provide common understanding between heterogeneous agents, systems, and organizations. The Web Services vision is of a world where loosely coupled, independently evolving (typically software) components. Several current efforts (OWL-S, SWSI, WSMO), are attempting to integrate the two visions, that is, to produce a world where Semantic Web ontologies supports greater automation of Web Services related tasks, such as service discovery and composition. For this purpose, the OWL-S [17] language was developed to provide a set of ontologies to describe services using the Web Ontology Language (OWL) [4].

Recently there has been a lot of work applying AI planning techniques to the Web Service composition problem. The straight-forward approach is to map service descriptions to planning operators and directly use existing planning systems. OWL-S allows

for describing services in ways amenable to planning. For example, it supports (in principle) describing the preconditions and effects of AtomicProcesses. Such AtomicProcess descriptions are easily treated as planning operators.

All existing versions of OWL-S have left the particular language for encoding preconditions and effects unspecified. Consequently, translation schemes from OWL-S to particular planning formalisms have had to insert their own encodings of preconditions and effects into the translated operators. As an unsurprising result, the translated precondition and effect formulas are easily handled by those planning systems. Unfortunately, the typical logic for expressing preconditions and effects in a planning system is quite differently expressive than RDF and OWL do. So, these systems are not exploring what it would be like to plan against actual encodings of world state that we expect to find on the Semantic Web. The forthcoming OWL-S 1.1 forces the issue by making the default language for encoding service preconditions and effects a variant of the Semantic Web Rule Language (SWRL) [10]. In order to evaluate such formulas planners must understand the semantics of OWL.

There are many likely impedance mismatches. For example, planners typically assume that they have the complete information about the world. Since it is assumed that planner knows all the objects and the relations, they use closed world reasoning with negation as failure. However, OWL has open world semantics because on the huge and only partially knowable World Wide Web a statement cannot be assumed true on the basis of a failure to prove it.

In this paper, we demonstrate how an OWL reasoner can be integrated with an AI planner to overcome these problems. The reasoner is used to store the world state, answer the planner's queries regarding the evaluation of preconditions, and update the state when planner simulates the effects of services. We first describe the challenges of modeling service preconditions and effects and world state using OWL, and then examine the impact of this on the planning process.

Specifically, we integrate the SHOP2 HTN planning system [16] with the OWL DL reasoner Pellet [18]. This work is an extension of our prior work for planning over OWL-S process models using SHOP2 [19]. In this work, we concentrate on the OWL DL fragment of the OWL language.

We also investigate the efficiency of such an integrated system and show how OWL reasoning can be optimized for this system. Finally, we present the performance results of our prototype implementation.

2 Preliminaries

2.1 Classical Planning Representation

In classical planning representation a state is a set of ground literals expressed in a first-order language. An action is an expression specifying which first-order literals must belong to the state in order for the action to be applicable, and which literals the action will add or remove in order to make a new world state. An atom p holds in state s iff $p \in s$. If g is a set of literals with variables, s satisfies g (denoted $s \models g$) when there is a substitution σ such that every positive literal of $\sigma(g)$ is in s and no negated literal of $\sigma(g)$ is in s.

In classical planning, a planning operator is a triple o = (*name*(o), *precond*(o), *effects*(o)). Effects of an operator can be positive or negative, i.e. *effects*$^+$(o) (generally referred as the add list) represents the set of literals that will be added to the state and *effects*$^-$(o) (generally referred as the delete list) represents the set of literals that will be removed from the state. An operator o is applicable in a state s when the preconditions are satisfied in the state, i.e. $s \models precond(o)$. Most planners represent the world state with a relational database and thus precondition evaluation is very fast. Applying the effects of an operator is done by adding or deleting entries from the database.

These definitions were based on the initial modeling of the STRIPS [6] system. Currently, widely accepted planning representations use more expressive precondition and effect descriptions [8]. For example, preconditions may contain disjunctions, quantified expressions and some form of axioms. The effects may be conditional and may also contain universally quantified expressions.

2.2 HTN Planning and SHOP2

HTN planning is similar to classical planning in that each world state is represented by a set of literals and each action corresponds to a state transition. However, 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.

Many service oriented objectives can be naturally described with a hierarchical structure. HTN-style domains fit in well with the loosely coupled nature of Web Services: different decompositions of a task are independent so the designer of a method does not have to have close knowledge of how the further decompositions will go. Such hierarchical modeling is the core of the OWL-S [17] process model to the point where the OWL-S process model constructs can be directly mapped to HTN methods and operators[19].

SHOP2 [16] is a domain independent HTN planner. A distinctive feature of SHOP2 is that it generates the steps of each plan in the same order that those steps will later be executed, so it knows the current state at each step of the planning process. This reduces the complexity of reasoning by eliminating a great deal of uncertainty about the world, thereby making it easy to incorporate substantial expressive power into the planning system. Thus SHOP2 can do axiomatic inference, mixed symbolic/numeric computations, and calls to external programs.

2.3 Description Logics

Description Logics are a family of class-based knowledge representation formalisms [1]. A DL knowledge base typically comprises two components: a “TBox” and an “ABox”. The TBox contains intensional knowledge in the form of a terminology and the ABox contains extensional knowledge that is specific to the individuals of the domain

of discourse. Intensional knowledge is usually thought not to change and extensional knowledge is usually thought to be contingent, or dependent on a single set of circumstances, and therefore subject to occasional or even constant change [1].

In DL implementations, core inference is typically the consistency check for ABoxes, to which all other inferences can be reduced. For example, checking if an individual a belongs to a concept term C in an ABox A can simply be done by checking if $A \sqcup \{a : \neg C\}$ is not consistent.

There is a direct correspondence between DLs and OWL. In fact, OWL DL and OWL Lite can be viewed as expressive Description Logics, with an ontology being equivalent to a Description Logic knowledge base. In particular, OWL facts (type assertions, property assertions, individual equality and inequality) corresponds to ABox assertions and OWL axioms (subclass axioms, subproperty axioms, domain and range restrictions, etc.) correspond to TBox knowledge.

2.4 Syntax and Notation

In our Web Service examples we will use a syntax similar to that of the Planning Domain Definition Language (PDDL) [8] since it is a middle point between the OWL-S surface syntax [15] and SHOP2's syntax. To express preconditions and effects we will use a syntax similar to N3 (see Figure 1). We will ignore the namespace prefixes for the URIs unless it is significant, e.g. rdf:type. Note that our service descriptions do not have output specifications but only input specifications. Since planning operators traditionally do not have outputs, OWL-S outputs are generally encoded as *knowledge effects* [19]. We will use the classical DL syntax ($\exists$, $\forall$, $\neg$) instead of verbose OWL names ($someValuesFrom$, $allValuesFrom$, $complementOf$) to describe concepts.

```
(:action register-course
 :parameters (?student - Student ?course - Course)
 :precondition (and (?course hasPrerequisite ?anotherCourse)
                    (?student passed ?anotherCourse))
 :effect       (?student registered ?course)
```

Fig. 1. A service that registers a student to a course. The precondition is that the student has passed the prerequisite course. The student is registered to the course as the effect of executing this service

3 Integrating an OWL Reasoner with a Planner

Integration of an OWL reasoner with a planner means that all of the planner's interaction with the state will be done by the reasoner. First, world state is actually represented as an OWL knowledge base. Any statement entailed by the KB is assumed to be true in the state. Evaluation of preconditions is done by the reasoner. Update to the state by the application of effects is also handled by the reasoner. The following sections explain the challenges of this integration. We do not discuss the soundness and completeness of the integrated system because it trivially follows from the fact that SHOP2 is sound and complete as long as its theorem proving is sound and complete.

3.1 Operator Definitions

We want to change the classical planning operator definitions such that preconditions and effects will be written with OWL. First we need to determine what kind of OWL statements can appear in operator preconditions and effects. For this purpose, we will look at what kind of formalisms have been used in planning community and how these can be used in our context.

The original STRIPS [6] language allowed the use arbitrary well-formed formulas in first-order logic for preconditions and effects. However, defining a semantics for this formulation was problematic [13]. Thus, in subsequent work, researchers have placed some restrictions on the nature of the planning operators.

Typically, preconditions and effects contain only first-order literals. This means that only SWRL atoms, which are in essence OWL facts (ABox assertions) with variables, can be used and we should exclude usage of arbitrary OWL axioms (TBox axioms) in operator definitions. This is also intuitive because the axioms in ontologies are used to model the world as we know it. They represent the nature of the world, e.g. student is always subclass of person, whereas the facts about individuals represent our current knowledge that may change over time, e.g. a person may graduate and no longer be a student.

Planners normally allow negated atoms to appear in preconditions. Planners generally operate with a closed world assumption and treat negation as failure. For example, a registration service may have a condition that only people who are not already registered may use that service and express this with the following precondition: not($?person$ rdf:type $Registered$). With NAF this would evaluate to true whenever we cannot prove the person is registered. However, with open world semantics failing to prove that the person is registered may just mean that we don't know if person is registered. To make sure that person is not registered, we want a stronger condition such as ($?person$ rdf:type $NotRegistered$) where $NotRegistered$ is the complement of $Registered$. As SWRL does not allow negated atoms appear in rule bodies, we also restrict the preconditions to contain only non-negated SWRL atoms.

One restriction planners impose on operator preconditions and effects is that only the variables defined as parameters can be used. It is easy to see that we cannot allow arbitrary variables to appear in effects because all literals we add to the state should be ground. However, this restriction can be relaxed as done in the Planning Domain Description Language (PDDL) [8] and implemented in expressive planning systems like SHOP. In particular, it is possible to use existentially quantified variables in the operator preconditions and universally quantified variables in the effects. When the variables in effects are universally quantified, we do not have the problem of unground variables because the variable will be bound to every instance in the state. The existentially bound variables in the preconditions may also appear in the effects as long as it is guaranteed that there will be only one substitution for that variable. If there is more than one substitution and planner chooses one of these options arbitrarily during planning all the rest of the plan may depend on this choice. Since there is no way of seeing this arbitrary choice in the plan generated (only the variables in the parameters can be known) there is no guarantee the same binding will be chosen during the execution of plan.

The restriction about variables do not apply to method preconditions. Since method descriptions in SHOP2 do not have any effects it is possible to use existentially quantified variables regardless of how many bindings for those variables may exist. Choosing a binding for this variable becomes a nondeterministic branching point for SHOP2. This feature is highly used in practice along with some heuristics about which bindings are most likely to yield a plan [16].

One problem about limiting use of variables in effects arises when the effect of an action is creating a new object that did not exist before. This problem emerges as a difficulty in modeling in some planning domains (see the Settlers domain in 2002 International Planning Competition [7]) and becomes ubiquitous when using OWL-S. Since OWL (and RDF) is based on triples, n-ary predicates must be described using some (possibly anonymous) intermediary individuals. These anonymous individuals, or so called bnodes, actually represent existential variables in the KB. Suppose the service description shown in Figure 2, which makes an appointment for a person with a doctor at a given time. Normally, this effect could be represented with a three variable predicate such as $appointment(?p, ?d, ?t)$. But using OWL requires us to define an additional object, i.e. $?appt$ variable, that will specify the relation between these three objects.

```
(:action make-appointment
 :parameters (?p - Person ?d - Doctor ?t - Time)
 :precondition ...
 :effect (and (?d hasAppointment ?appt)
              (?p hasAppointment ?appt)
              (?appt rdf:type Appointment)
              (?appt appointmentTime ?t)))
```

Fig. 2. A simplified service description where person $?p$ makes an appointment with doctor $?d$ at time $?t$.

These additional instances can be seen as the output of the service, i.e. the service creates a new appointment instance as an effect of its execution. But modeling these variables as outputs of the service would not be appropriate because output of a service is considered to be some data returned by the service after execution of the service. It is more proper to define a special category of variables to distinguish these"purely syntactic" variables from variables which are relevant to the planning problem. For example, in our implementation we used a simple syntax based solution where any variable that starts with a character '_' (as in Prolog don't care variables) is treated as an anonymous node rather than an existential variable.

Planners use axiomatic inference to infer conditions that were not in the world state. This extension establishes a distinction between two classes of predicates used in the domain: primitive and derived predicates. Derived predicates can be deduced from other primary and secondary relations whereas primary predicates are true only if they explicitly exist in the state. Including derived predicates in the effects of operators causes a problem as we will discuss in detail in Section 3.3. Commonly accepted solution to this problem is to allow only primitive relations to appear in effects of operators and restrict derived predicates to appear only in preconditions. This is quite an inconvenient restriction for OWL.

3.2 Precondition Evaluation

The applicability of a planning operator o in a state S is defined to be the satisfiability of its precondition in S. In other words, a planning operator is applicable if its precondition is the logical consequence of the state, written as $S \models precond(o)$. Preconditions are generally defined as conjunctions and since we have defined that preconditions can only contain OWL facts (or ABox assertions in DL terminology) possibly with variables, a precondition expression becomes equivalent to a conjunctive ABox query [11]. When the precondition expression does not contain any variables, precondition evaluation becomes boolean query answering, i.e. answering yes or no. When there are existentially quantified variables then we also need to generate the variable bindings that makes the conjunctive formula logical consequence of the state.

One important point in precondition evaluation is the presence of existentially quantified variables. The satisfiability of the preconditions actually depends on whether we want to get the variable bindings for these existential variables or not. This is a direct consequence of open world reasoning. Consider this simple example: Suppose we have a simple query ($?p$ hasChild $?c$). If we don't want to get the variable bindings for $?c$ then a KB containing only these assertions $\{Parent$ = $\exists$hasChild.$\top$, $John$:$Parent\}$ would satisfy the query with the binding $\{?p \leftarrow John\}$ because we know that $John$ has a child even though we do not know who that child is. On the other hand, when we want to bind the variable $?c$ to a known individual, the query would fail for the very same KB. The same behavior would be observed when there are anonymous individuals, individuals with no URI reference, in the KB.

Since the precondition evaluation highly depends on the interpretation of these existentially quantified variables we need to define a clear semantics as to which interpretation will be preferred. The OWL query language proposal [5] suggests to label the variables as *must-bind*, *may-bind*, and *dont-bind* to control this behavior. This is also consistent with ABox query answering schemes where some variables are labeled as *distinguished*, meaning they should be bound to a value.

Labeling the existential variables in preconditions as *dont-bind* variables cannot be done arbitrarily. A variable is *active* if it is used in another context, e.g. an operator may use it in the effects or a method may use it as an input of a subtask. An *active* variable should always be bound to a known individual to ensure that we always have ground terms. *Inactive* variables can be labeled as *dont-bind* or *must-bind* at will by service writer. It is preferable that an existential variable that is not labeled either way be interpreted as a *dont-bind* variable since this way we can benefit from the open world semantics of OWL to continue planning in the face of incompleteness in the KB.

As we have mentioned in section 2.1, current state of the art planning systems use more expressive constructs in preconditions such as disjunctions and quantified expressions. Evaluating a disjunctive would be equivalent to answering a disjunctive query. Note that answering disjunctive queries cannot simply be done by answering each disjunct separately because there are cases when the query itself is a logical consequence of the KB but none of its disjuncts are [11].

Universally quantified expressions in preconditions also creates a problem with open world semantics. Consider this simple precondition $\{(\forall\, x)(P$ hasChild $x)(x$:$Male)\}$ where it says that all the children of P should be male. The way planners evaluate quan-

tified expressions is with the closed world assumption where all the explicit children in the KB are found and tested with the condition. Then if we consider the following KB $\{ParentWithNoSon$=$\forall$hasChild.$Female$, $Female = \neg Male$, $John$:($\geq_1$hasChild $\sqcap$ $ParentWithNoSon)\}$ this closed world interpretation of the query would succeed although we know for sure that John has at least one daughter (again we just don't know who she is).

In most real world problems, preconditions involve some kind of numerical computation (i.e., comparison). It is foreseeable that a lot of services will use expressions such as the built-in primitives of SWRL to express these kind of preconditions. Consider the precondition of the book buying service shown in Figure 3. We can evaluate this precondition at two steps. In the first step, we do the query in our KB as described above and bind the variables $?price$ and $?limit$ to actual values. In the second step, we compare these two values and verify the condition holds. With this approach there are cases again where we can get incomplete results. Consider another condition where $\{(?p$ hasAge $?age)$, $(?age > 18)\}$ and a KB $\{PersonOlderThan40 = \exists$hasAge.$MoreThan40$, $John$:$PersonOlderThan40\}$ where $MoreThan40$ is defined as an XML Schema type with the restriction on its minValue to be greater than 40. In our KB, we don't have explicit information about John's age but we know that $\{?p \leftarrow John\}$ satisfies the condition (supposing $?age$ is a ***don't-bind*** variable). But the expressivity of OWL cannot handle more complex conditions, like the one in Figure 3, so it may be preferable to have another module that processes these expressions.

```
(:action buy-book
 :parameters (?b - Book ?cc - CreditCard)
 :precondition (and (?b hasCost ?price)
                    (?cc hasAvailableLimit ?limit)
                    (?price < ?limit))
 :effect ...)
```

Fig. 3. A simple book buying service where the available limit on the credit card should be higher than the price of the book

3.3 Applying Effects

The effects of an operator are applied to the current state to simulate the action. Applying an operator o to a state s transforms it into a new state denoted by $s_{new} = apply(o, s)$. After the application of effects, the atoms in the positive effects of the operator should be entailed by the state, i.e. $apply(o, s) \models effects^{+}\{o\}$, and the atoms in the negative effects should not be entailed, $apply(o, s) \not\models effects^{-}\{o\}$.

Applying the positive effects of an operator means adding new assertions to our KB which may cause inconsistencies. For example, a service may advertise a description where the effect of the service is given as ($?person$ president USA) saying that you will be the president of USA after running that service. However, if the current KB contains the information about the current president, i.e. there already exists another distinct individual who has the president property with value USA and the president property is defined as InverseFunctionalProperty, then adding this new assertion will cause an inconsistency. When there is an inconsistency in the KB any conclusion can be deduced so we cannot guarantee the correctness of the further results.

Most planners assume that modeling the planning operators correctly is the responsibility of the person who supplies the domain. The soundness and completeness of the planners are proven with respect to correct domain descriptions, e.g. a blocks world domain where an operator causes a block to be in two different places at the same time will cause most planners generate unsound plans. Since we are dealing with Web Service descriptions that come from various different sources we cannot guarantee the correctness of these descriptions. For this reason, a planner should reject the application of an operator when its effects cause an inconsistency.

Negative effects cannot cause an inconsistency in the KB because of the monotonic nature of our reasoning. Removing assertions from a consistent KB cannot cause it to become inconsistent. However, we have the problem of KB deriving the same assertion from other facts even after we remove that assertion from the KB. For example, an unregister service may have a negative effect which requires the deletion the fact ($?person$ member $Club$). But, if the KB includes another fact ($Club$ hasMember $?person$) such that hasMember is the inverse property of member then we will still derive the same conclusion as before. This is exactly why planning systems make the distinctions between primitive and derived predicates and do not allow derived predicates in effects (see section 3.1).

Unfortunately, restricting the usage of derived predicates in effects makes it nearly impossible to model any action in OWL. An OWL property p is a derived predicate if it satisfies any of the following conditions:

- It has a subproperty ($q \sqsubseteq p \equiv q(x,y) \rightarrow p(x,y)$)
- It has an equivalent property ($p = q \equiv q(x,y) \rightarrow p(x,y)$)
- It has an inverse property ($p = q^- \equiv q(x,y) \rightarrow p(y,x)$)
- It is a symmetric property (Symmetric $p \equiv p(x,y) \rightarrow p(y,x)$)
- It is a transitive property (Transitive $p \equiv p(x,y) \wedge p(y,z) \rightarrow p(x,z)$)

A type assertion in OWL such as (x rdf:type C) is equivalent to a single variable predicate in the form $C(x)$. This type assertion would be a derived predicate if class C meets any of the following conditions:

- It has a subclass ($D \sqsubseteq C \equiv D(x) \rightarrow C(x)$)
- It has an equivalent class ($C \sqsubseteq D \equiv D(x) \rightarrow C(x)$)
- It is defined to be the range of a property (p rdfs:range $C \equiv p(x,y) \rightarrow C(y)$)
- It is defined to be the domain of a property (p rdfs:domain $C \equiv p(x,y) \rightarrow C(x)$)

Note that being a subclass of some restriction could also cause C to be a derived predicate, e.g. $D \sqsubseteq \forall p.C \wedge D(x) \wedge p(x,y) \rightarrow C(x)$. It is even hard to enumerate all these case because the combination of cardinality restrictions, nominals and general inclusion axioms may cause class membership to be derived from other facts.

If we allow derived predicates to appear in negative effects then we need a way to make sure that statement will not be inferred after the effect is applied to the world state. One possibility is to make the reasoner delete all the related statements from the KB until the statement in question is not entailed by the KB. Given the expressivity of OWL DL this is quite a hard task. Furthermore, there is no deterministic way to control this behavior. For example, in the KB $\{x{:}A,\ x{:}B\}$ if we want to delete $x{:}A \sqcap B$ then

we can either delete x:A, x:B or both to have the same effect. Another possibility is to make the service writer include all the enumerations, other predicates that the truth value depends on, in the negative effect list. This works well for simple domains but gets quite hard quickly when the ontologies and definitions become complex. It is even harder in the distributed setting of the Web where a service writer may enumerate all the possibilities in the description to the best of her knowledge but the client who uses that description may have access to another ontology that augments those definitions with some new descriptions with dependencies not mentioned in the negative effects.

4 Implementation and Optimization Techniques

The performance of the planning system is considerably affected when the precondition evaluation of operators and methods are done by theorem proving. During a plan generation, planner will do hundreds of precondition evaluations so the reasoner needs to handle these queries very fast to be at all workable.

A significant majority of the preconditions consist of conjunctive expressions so we will focus on how to optimize conjunctive queries. As we have discussed in section 3.2, operator preconditions (generally) do not contain variables whereas method preconditions have many existentially quantified variables. If the precondition does not contain any variables we just need a yes/no answer, whereas the preconditions with *must-bind* variables then we have to generate answer sets for these variables.

The existing conjunctive ABox query answering algorithms [11, 12] reduce the problem of query answering to one or more KB satisfiability problems. The main idea is to consider a conjunctive query as a directed graph where the nodes are either variables or individual names (constants). In addition, concept and role terms provide labels for nodes and edges respectively. For example, the query $\{(?x$ rdf:type $Start)$, $(?x$ path $?y)$, $(?z$ path $?x)\}$ corresponds to a graph with three nodes and two edges. When the query consists of one connected graph then the query can be answered with one satisfiability test.

Answering queries with only one term, i.e. the query graph has no edges, is equivalent to an entailment check. For example, the query (A rdf:type $Rover$) is entailed by the KB S if and only if $\{S \sqcup (A$ rdf:type $/Rover)\}$ is not consistent. When the query contains multiple terms, i.e. the query graph has more than one edge, then the technique of "rolling up" is applied to transform the query into an equivalent query with a single concept term. For example, the following query that has no variables $\{(C$ rdf:type $Computer)$, $(C$ manufacturedBy $M)$, $(C$ hasCPU $CPU)$, $(CPU$ cpuType $Centrino)\}$ can be transformed into the following concept term (C:$\exists$.manufacturedBy $\{L\} \sqcap \exists$hasCPU. $(\{CPU\} \sqcap \exists$cpuType.$\{Centrino\})$). The query can now be answered by adding the negation of this concept to the individual A and then checking if the KB is consistent. If the query contains multiple disconnected components, each connected subcomponent can be rolled up to one individual and tested separately.

Rolling up technique is quite effective when we don't need the variable bindings because one query that contains multiple terms can be answered with one satisfiability check rather than multiple entailment tests. However, this technique is not efficient when we also want the variable bindings. The variable bindings are returned by replac-

ing each variable with one individual, rolling up the query and answering the boolean query. One must try every possible combination of bindings to get all the answers. [12] proposes an optimization technique that attempts to reduce the number of candidate individuals. The idea is to roll-up the query into a distinguished variable prior to substituting it with any individual name. The concept is used to retrieve the list of individual names corresponding to instances of the concept. The retrieved individuals are used as the candidates for the distinguished variable.

This technique reduces the number of satisfiability tests but still tries unnecessary tests. Consider the previous query with all the individual names are replaced with variables $\{(?c$ rdf:type $Computer)$, $(?c$ manufacturedBy $?m)$, $(?c$ hasCPU $?cpu)$, $(?cpu$ cpuType $?t)\}$ where we want to get all the computers, their manufacturers, the CPU they have and the type of these CPUs. Suppose we have 10 computers manufactured by 10 different manufacturers and each computer has only one CPU (for a total of 10 distinct CPU instances) and three types of CPUs, Pentium3, Pentium4 and Centrino. In the original setting, we need to try each individual. Since we have 33 individuals, assuming nothing else exists in the world, we could try every combination of bindings for a total of $33^4 \approx 1186000$ consistency tests. The optimization described above would help us to reduce the number of candidates so we would not try to use a manufacturer as a candidate computer. Therefore, we have 10 different possibilities for variables $?c$, $?m$, $?cpu$ and 3 candidates for $?t$. The algorithm still tries all possible combination of these bindings yield a total of $10 \times 10 \times 10 \times 3 = 3000$ tests.

The problem with this approach stems from not having the ability to see why a binding fails. For example, if computer $C1$ is manufactured by $M1$ then a binding with $C1$ and $M2$ will fail no matter what candidates we try for the other variables. Unfortunately, it is not possible to learn the dependencies between variable bindings using the rolling up technique. For this purpose, we propose a new technique where each individual term in the query is tested separately as an entailment test. For the given query example, given a candidate binding for a computer we would try the 10 different manufacturers and find the one binding that is the logical consequence of the KB. Then we would try 10 different CPU bindings, out of which only one succeeds. Then we try the remaining 3 candidates for the CPU types. In the end, we end up trying only a total of $10 \times (10 + 10 + 3) = 230$ consistency tests.

Computing the likely candidates itself is a costly operation. In the example query we have four distinguished variables so we need to perform four instance retrieval operations. Generally, reasoners realize the whole KB upon loading and this retrieval operations become cheap. Unfortunately, in our setting planner is constantly changing the current state possibly invalidating the cached results. It is much preferable to use the optimized instance retrieval algorithms designed for dynamically changing ABoxes [9]. The motivation of this approach is to eliminate all of the irrelevant individuals with only one consistency check. Obvious instances of the concept need not be tested at all and the rest of candidates can be eliminated with a binary partioning method. The idea for retrieving the instances of concept C is to add $\{x{:}\neg C\}$ assertion for every x that cannot be eliminated by inspection. If the new KB is consistent we conclude that no more instances of C exist in the remaining set, otherwise KB is partitioned to half and

this procedure is continued at each partition. Thus, at each step binary partitioning may eliminate half of the candidates using a single test.

Computing the candidates by rolling up the whole query gives too many possibilities. If we compute the candidates based on each statement and the bindings done at previous steps then we will find a smaller number of candidates that are more likely to succeed at later steps. When we concentrate on the statements of the query we can also make use of the existing assertions in the KB more efficiently. In most DLs looking at the existing role assertions is enough to determine if two individuals are related to each other with a given role. However, in the presence of nominals this is not the case any more and we may get incomplete results with this approach. But we use structural inspection to find obvious answers and then use optimized retrieval on the rest of the individuals we can get complete results efficiently. For example, if the statement in the query is ($?s\ p\ o$) we can first examine the existing role assertions to get the obvious answers. Then we can retrieve the instances of the concept $\exists p.\{o\}$ to get the remaining bindings for $?s$. Note that, if all the individuals are related with explicit assertions then only one consistency check (as described above) will be enough to eliminate all the other possibilities.

When combined with an iterative query answering mechanism this approach may help to avoid a lot of consistency tests. In a planning problem, most of the time, finding the first plan is enough (e.g. if we are not trying to optimize a cost function). In this case, we can first try the obvious candidates and delay the consistency test as much as possible. If the planner cannot find a plan with the initial bindings then it would keep asking the reasoner for more bindings which in the end would require us to make an expensive consistency test. But there is a good chance that a plan can be found with the initial bindings.

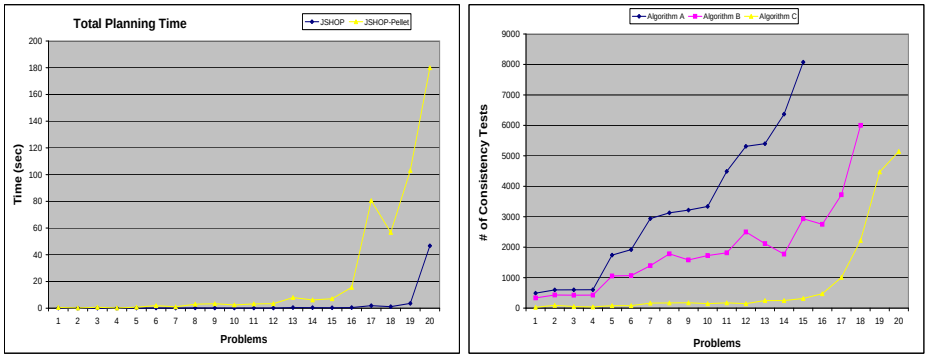
5 Experiments

We have done some experiments (1) to evaluate the optimization techniques we described and (2) to compare the performance of the integrated system with the original planning system. We have built a prototype system by integrating the OWL DL reasoner Pellet [18] with the Java version of SHOP (JSHOP). We ran our experiments on a Windows machine with a Pentium Centrino 1.6 GHz CPU and 1GB memory.

Since there are no standard benchmark problems for service composition, we have done our experiments with the Rover domain which was used in the 2002 International Planning Competition [7]. In this domain a collection of rovers navigate a planet surface, find samples and communicates the results back to a lander. We have translated 20 problem files to OWL and encoded the original JSHOP domain in our syntax where precondition and effects are written as SWRL atoms. Since this domain was using n-ary predicates, e.g. $can\_traverse(rover, loc1, loc2)$, we had to translate them using extra individuals, e.g. ($rover$ can_traverse $path$, $path$ begins $loc1$, $path$ ends $loc2$). For this reason, OWL versions of problems were slightly different than the original, especially the harder problems contained significantly more individuals.

We first implemented the three different query optimization mechanisms and ran them on the test cases. Figure 4(a) shows the results of our experiments. Algorithm A is

the original optimization technique as suggested in [12]. It computes the candidates for each variable by rolling up the query to that term and performing the binary instance retrieval algorithm. The query then is rolled up for each combination of the candidates. Algorithm B also computes the candidates in the same manner but uses the variable dependencies to prune the possibilities that will obviously fail. Algorithm C on the other hand does not compute the candidates in the beginning. Candidates are computed on a need basis. As some of the other variables are bound to actual values, the candidates for the remaining variables are computed using these values. The results show Algorithm C always performed significantly better than the others. Note that Algorithm A and B was unable to finish all the test cases even after a substantial amount of time. These are still preliminary results that need to be confirmed with a wider variety of KBs.



(a) **(b)**

Fig. 4. Chart (a) shows the comparison of the total planning time spent by the original JSHOP system and the integrated JSHOP-Pellet system. Chart (b) shows the number of consistency tests done by each variation of the optimization methods.

We then compared the performance of the integrated system with the original JSHOP system. As we have expected original JSHOP performed better in every test case as shown in Figure 4(b). But we should also note that Pellet is not highly optimized to handle large number of individuals. In our experiments, we also tried to use Racer which is highly optimized for such KBs. However, only way to communicate with Racer was through HTTP sockets which dominated the total time spent and in the end overall performance was not any better.

6 Related Work

The most closely related work to ours is McDermott's Optop planner [14]. Optop is an estimated-regression planner that generates compositions of Web Services where the goal is given as a logical formula. This approach is applicable to atomic service descriptions where we use HTN planning to deal with composite service descriptions. Another difference is that, in our work, we have focused on how to deal with the expressivity of OWL ontologies and do reasoning during planning whereas Optop employs Horn-logic axioms for inference.

There has been some work to combine Description Logics with planning but with a completely different approach. In [3] De Giacomo et al. suggests using a DL framework to plan for robot actions. This approach uses the formalization of actions given by propositional dynamic logics (PDL). The correspondence between PDLs and DLs are exploited for an actual implementation using the reasoner CLASSIC. Badea [2] also makes use of this correspondence and presents two deductive approaches, where the existence of a plan corresponds to an inconsistency proof, as well as a satisfiability based one, where planning is reduced to model construction. Both approaches use PDL framework to represent actions whereas we use DL formalism to represent states.

Query answering has been investigated in a DL framework [11] and in the Semantic Web context [12]. In our work, we have used the sound and complete algorithms presented there and concentrated on optimizing the query answering time. Our main focus was to minimize the number of consistency tests without changing the main algorithm and treating the consistency test as a black box operation.

7 Conclusions and Future Work

In this work we have investigated the issues of using planning for composition of Web Services on Semantic Web. We examined the impact of using OWL to describe the preconditions and effects of services. We have shown what features of classical planning are not suitable in Semantic Web. In particular, we identified the challenges to write service descriptions and reason about them with open world semantics and the expressivity of OWL. These issues need to be addressed in order to develop real-world applications on Semantic Web.

We have shown how an OWL reasoner can be coupled with a planner to reason about the world state during planning. The efficiency of such a system has been investigated and we presented novel optimization techniques that would be useful for query answering in general. Our preliminary experiments with the prototype implementation showed that the performance of planning system suffers due to the characteristics of DL reasoning. But it is also clear that there are plenty of optimization possibilities that need to be investigated.

Querying large Semantic Web KBs using Description Logic based algorithms will most likely become an important and popular topic. Optimization techniques for query answering is anticipated to get more attention in this context. As a future work, we aim to evaluate our optimization techniques more thoroughly using ontologies of varying size and structure. We also think that examining the inconsistency results based on clash dependencies could help us to speed up the elimination of candidates that cause us to repeat the consistency tests.

In our experiments, we have used domains coming from the planning literature. As a future work, we will concentrate on scenarios related to actual Web Service composition problems. As the new version of OWL-S becomes available we expect to find more real world examples that we can use in our system.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logics Handbook: Theory, Implementations, and Applications*. Cambridge University Press, 2003.
2. L. Badea. Planning in description logics: Deduction versus satisfiability testing. In *European Conference on Artificial Intelligence*, pages 479–483, 1998.
3. G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Classic planning for mobile robots. In *Proceedings of the FAPR-96 Workshop on Planning in Complex Environments*, 1996.
4. M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web Ontology Language (OWL) Reference. W3C Recommendation 10 Feb 2004 http://www.w3.org/TR/owl-ref/.
5. R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Technical Report KSL-03-14, Stanford University, CA, 2003.
6. R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 88–97. Kaufmann, San Mateo, CA, 1990.
7. M. Fox and D. Long. International planning competition, 2002. http://www.dur.ac.uk/d.p.long/competition.html.
8. M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains, 2002. http://www.dur.ac.uk/d.p.long/pddl2.ps.gz.
9. V. Haarslev and R. Mller. Optimization strategies for instance retrieval. In I. Horrocks and S. Tessaris, editors, *Proceedings of the 2002 Description Logic Workshop (DL 2002)*, volume 53 of *CEUR Workshop Proceedings*, 2002.
10. I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM, 2004.
11. I. Horrocks and S. Tessaris. A conjunctive query language for description logic aboxes. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI 2000)*, pages 399–404, 2000.
12. I. Horrocks and S. Tessaris. Querying the semantic web: a formal approach. In *Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002)*, 2002.
13. V. Lifschitz. On the semantics of strips. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans*, pages 1–9. Kaufmann, Los Altos, CA, 1987.
14. D. McDermott. Estimated-regression planning for interactions with web services. In *Sixth International Conference on AI Planning & Scheduling*, 2002.
15. D. McDermott. Surface syntax for OWL-S, 2003. http://www.daml.org/services/owl-s/1.0/surface.pdf.
16. D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.
17. OWL Services Coalition. OWL-S: Semantic markup for web services, 2003. OWL-S White Paper http://www.daml.org/services/owl-s/0.9/owl-s.pdf.
18. Pellet. Pellet - OWL DL Reasoner, 2003. http://www.mindswap.org/2003/pellet.
19. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, October 2003.