Automated Composition of Web Services using AI Planning Techniques

Evren Sirin
Department of Computer Science,
University of Maryland,
College Park, MD, 20742
evren@cs.umd.edu

Abstract

Web Services is an emerging paradigm in which very loosely coupled software components are published, located, and invoked on the Web as parts of distributed applications. Web Services provide a new way of distributed computing where the interoperability between diverse applications is achieved through platform and language independent interfaces. The main focus of Web Services is the ability to easily combine existing components to create compositions that provide novel functionality that was not directly available from the existing services. Web Services composition is useful for a wide range of audience: ordinary users doing everyday tasks on the Web, commercial organizations involved in e-business applications, and researchers doing intense scientific computation over distributed networks such as the Grid.

Automated composition of Web Services requires fairly rich machine-understandable descriptions of services that can be shared between heterogeneous agents. Given appropriate descriptions, AI planning techniques can be employed to automate the composition of Web Services described this way. However, Web Service Composition problem differs from classical planning problems in various ways. The information about the world is incomplete and constantly changing; the domain knowledge, i.e. Web Service descriptions, have been developed by different parties and are distributed over the Web; and the plans generated involve communication and interaction with other agents, i.e. providers of the Web Services. In this setting, creating a composition to accomplish a goal requires to interleave planning with execution where both the state of the world and the domain knowledge about planning operators are gathered during the planning process.

In this paper, I present the preliminary work I have done for automating the composition of Web Services and discuss future directions for overcoming the limitations of the preliminary work. The purpose is to show that AI planning techniques can be extended to automatically generate useful and purposeful compositions of Web Services under incomplete information. As a starting point, I have worked on how Web Ontology Language (OWL) can be used to describe Web Services. I have created an interactive tool for a user-oriented composition approach and also studied how HTN planning can be used for automated composition of Web Services. In both systems reasoning with Web Ontologies has been used to facilitate the composition task. The goal of this paper is to investigate how these approaches can be combined, with each other and can be extended to efficiently address the issues related to the nature of the Web, i.e. a large, distributed, dynamic environment with incomplete and possibly inconsistent information.

1 Introduction

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services are designed to provide interoperability between diverse applications. The platform and language independent interfaces of web services allow the easy integration of heterogeneous systems.

Service Oriented Architectures tend to be component oriented with loose coupling as a systematic design emphasis. Services should not only be loosely coupled with their implementation, but they should be able to be coupled together with a minimum difficulty so that *combinations* of services can be separated from their particular realization. Given such combinations — called service *compositions* — a service consumer can mix and match components at will depending on service availability, quality, price, and other factors.

While realizing service compositions on particular concrete services is an important task, *generating* such compositions to achieve new functionality is equally or sometimes even more important. Creating novel functionality by means of composition is essential when there is no single service capable of performing that task but there are combinations of existing services that could.

There are many different application areas where automatic composition of Web Services would be useful.

- *Web Tasks* There are many tasks ordinary users perform on the Web everyday. When the objective requires interacting with different parties – e.g. making travel arrangements may involve buying plane tickets, booking hotel rooms, and renting cars – the task becomes tedious. Locating the services with required specifics and coordinating the flow between these sources is not an easy task.

- *B2B Applications* The composition is very important in B2B applications where online partnerships can automatically be formed without prior agreements. A business who wants to order some items from a manufacturer and then arrange the shipment details can achieve this goal by combining the services provided by manufacturers and shipment companies. These compositions allow the formation of dynamic trading communities.

- *Grid Applications* The Grid provides a computational framework to solve large-scale problems in science, engineering, and commerce. Many tasks on the Grid requires the coordination and combination of multiple services and resources. Composing these services in workflows of varying complexity is required for different tasks. For example, a scientist working on bioinformatics would want to get data about DNA, apply some specific tests on the data and then transform the results to a certain format. Each of these services may be provided by different sources and need to be combined together to satisfy the goal.

- *Pervasive Computing* Exposing the functionality of devices as Web Services provides a uniform method for describing the capabilities of these devices and thus enables us to compose these services together. In today's device-rich environment, most tasks require the ability to compose services such as the ones provided by printers, projectors, kiosks with the programs on your computer or the services available on the Web.

All these examples come from relatively different areas but still share some fundamental characteristics:

- *Distributed Setting* Service descriptions are created by different sources that do not necessarily share common knowledge or understanding. This implies that services that will be used to create the composition should be discovered from remote sources. This discovery process should take into consideration possible misalignments between the vocabularies of Web Service descriptions.

- *Incomplete Information* The composition system will have incomplete information about the world. When the size and nature of Web is considered, we cannot assume that the system already knows the information needed to find a composition. As the set of services grows very large (i.e., as we start using large repositories of heterogeneous services) it is likely that trying to complete the initial state will be wasteful at best and practically impossible in the common case.

- *Interleaved Execution and Composition* The composition system should execute the necessary information-providing services during the composition process to gather information. While not all the information relevant to a problem may have already been known, i.e. the amount of money in the bank account, it will often be the case that that information is accessible to the system, i.e. by using the service provided by bank's Web site. The relevance of possible information can be determined by the possible combinations the planner is considering, so it makes sense to gather the information at that point.

- *Web Scale* A system to compose Web Services should scale to the Web standards where the number of available services may be in the order of millions. It is not possible to handle this number with any naive approach.

The dynamic composition of services primarily requires understanding the *capabilities* of the available services (i.e., *what* they can do) and the *compatibility* of those services. Several technologies, such as SOAP [45], WSDL [10], [44], are being developed to provide a standard way of describing Web Services. However, Web Service standards mainly concentrates on the syntactic properties of the descriptions , i.e. syntax of the descriptions, structure of messages exchanged between services, etc. Automating the composition process requires more comprehensive descriptions where the semantics of a Web Service can be expressed in a machine-understandable format. The means for sharing information between separate parties needs to be established in order to combine the services together to achieve the overall goal of the composition.

The Semantic Web vision is of a world where loosely coupled, independently evolving ontologies provide common understandings between heterogeneous agents, systems, and organizations. Several current efforts (OWL-S [34], SWSI [1], WSMO [13]), are attempting to integrate the two visions, that is, to produce a world where Semantic Web ontologies support greater automation of Web Service related tasks, such as service discovery and composition.

Fairly rich Web Service descriptions provide the means to understand the semantics of single services but it will still be required to put these services together to accomplish goals that cannot be simply fulfilled with an individual service. Producing a sequence of actions to reach a certain goal is the objective in AI planning. Web Service descriptions can be mapped to action definitions and an AI planner can be used to generate compositions of Web Services.

The purpose of my research is to show that AI planning techniques can be extended to automatically generate useful and purposeful compositions of Web Services under incomplete information. My work will provide a basis for how to encode service descriptions of sufficient richness to support partial to full automation of composition.

As a starting point, I have worked on how Web Ontology Language (OWL) can be used to describe Web Services and their capabilities. This work resulted in developing tools that partially automates of generating expressive Web Service descriptions and a composition tool that helps users by using these descriptions to do selection and filtering of the services. I have developed a Description Logic based OWL reasoner that was used to find combinations of Web Services and filter the results based on user constraints. As a preliminary work on automated composition I have worked on mapping Web Service descriptions written from OWL-S to SHOP2 planning domains. I have focused on the incomplete information problem and the issues related to interleaving execution with planning process. I have also examined how to handle the expressivity of ontologies in the planning process. I have extended my work on reasoning procedures to effectively handle precondition evaluations of the planner when the knowledge about the state of the world is expressed in OWL.

In my future work, I will primarily concentrate on how to address the issues and challenges listed above. I will work on extending the existing HTN planning paradigm to be able to generate plans with operators and methods that are described by separate sources. Enriching the task and method representation to allow easy-sharing descriptions between different parties while improving the ability to match remote methods with tasks at hand is going to play an important role. I will develop algorithms and methodologies to interleave planning with execution not just to gather information about the state of the world but also discover new planning operators and augment the domain knowledge about the problem. I will not limit myself to use Web Services described in a planning-oriented language such as OWL-S but also try to extend the horizon to use less expressive Web Service descriptions, e.g. Web Services that do not have explicit precondition/effect specifications but be associated with a message exchange patterns or classified in a Web Service taxonomy.

2 Background and Related Work

2.1 Semantic Web and Ontologies

2.1.1 Semantic Web Languages

The Semantic Web [6] is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. This is realized by marking up Web content, its properties, and its relations, in a reasonably expressive markup language with a well-defined semantics.

Semantic Web languages are used to represent information about resources on the Web. This information is not limited to be about Web resources but can be about anything that can be identified. Uniform Resource Identifiers (URIs) are used to uniquely identify entities. For example, it is possible to assign a URI to a person, to the company he works for, to the car he owns, etc. so relations between these entities can be written and shared on the Semantic Web.

There is a stack of languages that have been published as W3C recommendations to be used on Semantic Web. At the bottom layer of the stack, there is the Resource Description Framework (RDF) [9]. RDF is a simple assertional language that is designed to represent information in the form of triples. Triples are statements that contains a subject, a predicate and an object. RDF Schema (RDFS) [8] is a collection of RDF resources that can be used to describe properties of other RDF resources. Unlike its name suggests, RDFS is not a schema that specific constraints on the structure of an document, but instead provides information about the interpretation of the statements given in an RDF data model. In this regard, RDFS has similarities to frame based languages and can even be described as a relatively inexpressive Description Logic (DL). Though it should be noted that RDFS has a much more free representation and quite different semantics that traditional DLs.

The Web Ontology Language (OWL) [14], is the most expressive standardized Semantic Web language that is layered on top of RDF and RDFS. OWL can be used to define *classes* (unary relations) and *properties* (binary relations) as in RDFS but also provides constructs to create new class descriptions as logical combinations (intersections, unions, or complements) of other classes, define cardinality restrictions on properties and so on. OWL has three different species: OWL Lite, OWL DL and OWL Full. OWL Lite and DL differ from OWL Full such that they define certain constraints on RDF and RDFS so as to be compatible with the traditional semantics of DLs.

2.1.2 Reasoning on Semantic Web

The semantics of unrestricted RDF-S and OWL Full is non-traditional and the reasoners built for OWL Full fragment tend to be sound but incomplete. Since there is no straight-forward way to extend the existing reasoners to support the full expressivity of OWL Full. Therefore, focusing on OWL DL fragment of the language and use the sound and complete reasoning techniques developed for Description Logics.

Description Logics are a family of class-based knowledge representation formalisms [4]. 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 [4].

In DLs, there is one main inference problem, namely 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. Almost all other reasoning tasks, i.e. entailment, query answering, can be reduced to consistency checking.

2.2 Web Services

2.2.1 Web Service Standards

There are various different standards that have been developed for different Web Service tasks such as description, discovery and invocation. These technologies are primarily designed to be used in conjunction with other Web standards, e.g. XML for syntax and HTTP for communication.

SOAP [45] is the communication protocol designed to exchange messages between applications over the Web. It is fundamentally a stateless, one-way message exchange paradigm, but applications can create more complex interaction patterns by combining such one-way exchanges. SOAP provides a distributed processing model where a SOAP message is delivered from a sender to an ultimate receiver via zero or more SOAP intermediaries. This distributed processing model can support many message exchange patterns including but not limited to one-way messages, request/response interactions, and peer-to-peer conversations.

Web Service Description Language (WSDL) [10] is the language to describe the mechanics of interacting with a particular Web service. The abstract functionality of the Web service is defined in terms of the types of messages it sends and receives in WSDL *interface*. An interfaces is a set of *operations* and an operation is a sequence of input and output messages. An operation associates a message exchange pattern (MEP) with the message types that will be exchanged in that operation. The message types are defined using a schema language such as (but not limited to) XML Schema. The abstract interfaces are associated to concrete message formats and transmission protocols with *binding* descriptions.

Universal Description Discovery and Integration (UDDI) [44] is an emerging standard registry system for Web Services. UDDI allows businesses to advertise their Web Services by publishing their descriptions on a global registry. There are three main parts of this registry: White Pages that list contact information about the company that developed the Web service; Yellow Pages that organize Web services by such categories as geography and industry code; and Green Pages that hold WSDL descriptions. UDDI supports the association of an unbounded set of properties to the description of Web Services via a construct called TModel. For example, a service may specify its category using an arbitrary classification system though their meaning is not codified, therefore there may be two different TModels with the same meaning, but this similarity cannot be recognized.

Business Process Execution Language for Web Services (BPEL4WS) [12] is a language to define compositions of Web Services. It uses WSDL descriptions as the building block of the composition. BPEL4WS process is a kind of a flow-chart composed of *activities*. An activity is either a *primitive* or a *structured* activity. Primitive activities include single step operations such as invoking an operation on some Web Service or waiting for a message from an external source. Structured processes defined compositions using constructs such as sequences, conditionals, loops and so on. Similar to WSDL, BPEL4WS allows both abstract (not executable) and concrete (executable) descriptions. An abstract process description specifies the message exchange behavior between different parties without revealing the internal behavior for any one of them. An executable process, on the other hand, specifies the execution order between a number of activities constituting the process, the partners involved in the process and the messages exchanged between these partners.

OWL-S [34] provides a set of OWL ontologies to describe Web Services in a more expressive way than allowed by WSDL. The features of the Web Service, e.g. message types, constraints and capabilities, are defined using the terms from Web Ontologies. OWL-S partitions the semantic description of a web service into three components: the service profile, process model, and grounding. The *ServiceProfile* describes what the service does by specifying the input and output types, preconditions and effects. The *ProcessModel* describes how the service works; each service is either an *AtomicProcess* that is executed directly or a *CompositeProcess* that is a combination of subprocesses (i.e., a composition). The *Grounding* contains the details of how an agent can access a service by specifying a communications protocol, parameters to be used in the protocol, and the serialization techniques to be employed for the communication. The similarities between OWL-S and other technologies may be briefly expressed as follows. The ServiceProfile is analogous to yellow-page- like advertisements in UDDI, the ProcessModel is similar to the business process model in BPEL4WS, and the Grounding is a mapping from OWL-S to WSDL. The main contribution of OWL-S is the ability to support richer descriptions of the services and the real world entities they affect in such a way as to support greater automation of the discovery and composition of services.

Both BEPL4WS and OWL-S represents compositions from the perspective of a single party. The client is responsible from handling the control and data flow between the components of the composite service. This view of composition is called *orchestration*. This view differs from *choreography*, which is more collaborative in nature and aims to describe each involved party's part in the interaction so each participant will exact know how to interact with others. The choreography description outlines the roles of participants, their obligations in the choreography, and the order and structure of messages exchanged between these participants. A party who wants to participate in this choreography needs to obey these rules. A W3C working group is now developing the Choreography Description Language (CDL) to specify the details of such a description language.

2.2.2 Web Service Discovery and Matching

Research on Semantic Web Service discovery and matching has primarily focused on using the subsumption relation between Web Service advertisements and requests. And more specifically the subsumption relation between the input and output types have been used to generate matchings for Web Services that were defined using OWL-S. The DAML-S [1] Matchmaker [35] is the first system that implemented this idea is a system. The Matchmaker is designed to augment the current UDDI architecture with semantic service descriptions. Using concepts from Web Ontologies for matchmaking allows to find flexible matchings beyond the capabilities of UDDI which only supports text based matching.

The Matchmaker system uses DAML-S (or, presumably in a successor, updated Matchmaker, OWL-S) profiles to describe service requests as well as the services advertised. A service provider publishes a DAML-S description to a common service repository. When someone needs to locate a service to perform a specific task, a ServiceProfile for the desired service is created. Request profiles are matched by the service registry to advertised profiles using DL subsumption as the core inference service. In particular, the Matchmaker computes subsumption relations between each individual input, output, precondition and effect (IOPE) of the request and the advertisement ServiceProfile. If the classes of the corresponding parameters are equivalent, there is an exact and thus best match. If there is no subsumption relation, then there is no match. Given a classification of the types describing the IOPEs, the Matchmaker assigns a rating depending on the number of intervening named classes between the request and advertisement parameters. Finally, the ratings for all of the IOPEs are combined to produce an overall rating of the match. In summary, the basic rating used in matchmaking are as follows:

- **Exact** If advertisement A and request R are equivalent concepts, it is called an Exact match
- **PlugIn** If request R is sub-concept of advertisement A, it is called a PlugIn match
- **Subsume** If request R is super-concept of advertisement A, it is called a Subsume match
- **Fail** Otherwise, there is no match

[21] and [29] extends the matchmaking algorithms to exploit more features of subsumption relations. For example, when there is no subsumption relation between the advertisement and request, a rating called *Intersection* may be assigned when their intersection is not empty, i.e. advertisement and request descriptions are not disjoint. This case implies that relaxing some of the constraints on the request may provide better results. And both approaches differ from the Matchmaker because they use the whole service description, or more correctly the profile description, for discovery purposes and try to find the subsumption relation between these more complex class expressions. Lei and Horrocks point out a problem about OWL-S profile descriptions where encoding too much information in the profile, e.g. name and address of the provider, prevents effective matching. They overcome this problem by separating various components of the description; in particular the description of the service being provided was separated from the descriptions of the providing and requesting “actors”.

[1]OWL-S was formerly named as DAML-S and was based on DAML+OIL

Benatallah et al. [7] presents a different matching algorithm, called "best profile covering problem" to support flexible matching beyond equality and subsumption matches. In this approach, matching the service request is encoded as a new instance of the problem of rewriting concepts using terminologies. The goal is to rewrite a Web Service request description R into the closest description expressed as a conjunction of (some) concept names (Web Service descriptions) in an ontology O. To enable flexible matchings, a difference operation on service descriptions is proposed to meet this requirement. Such an operation enables to extract from a subset of Web service descriptions, the part that is semantically common with a given service request and the part that is semantically different from the request. Knowing the former and the latter allows to select relevant Web services and then to choose the best ones.

Roughly speaking, the difference of two descriptions C and D, expressed using $C - D$, is defined as being a description containing all information which is a part of the description C but not a part of the description D [43]. However, it is worth noting that, in some description logics, $C - D$ may be a set of descriptions which are not semantically equivalent. Teege [43] provides sufficient conditions to characterize the logics where the difference operation is always semantically unique and can be implemented in a simple syntactical way by computing the set difference of subterms in a conjunction. According to [43], structural subsumption is a sufficient condition that allows to identify such logics.

In the profile cover algorithm, difference operator is applied to the inputs (denoted by $I(R)$) and outputs (denoted by $O(R)$) of a request profile R and advertisement profiles. Finding a set of advertised profiles A to minimize $O(R) - O(A)$ ensures that the resulting set will satisfy the required output constraints. The algorithm considers both the *missing* and the *extra* information in the found answer set. The implementation of the algorithm is done by computing the minimal cost transversals of a hypergraph.

2.2.3 Automated Web Service Composition

Narayanan and McIlraith [32] define the semantics for a relevant subset of DAML-S in terms of the situation calculus. Atomic process descriptions, preconditions and effects in DAML-S are mapped to situation calculus constructs. McIlraith and Son [31] extends this mapping to encode composite processes in Golog [28], a high-level logic programming language built on top of the situation calculus. They adapt and extend the Golog language to enable programs that are generic, customizable and usable in the context of the Web. To support information gathering combined with search, they propose a middle-ground Golog interpreter that operates under an assumption of reasonable persistence of certain information. A ConGolog interpreter is augmented with online execution of information-providing services with offline simulation of world altering services.

Berardi et al. [5] presents a framework in which the exported behavior of a Web Service is described in terms of its possible executions (execution trees). The framework is specialized to the case in which such exported behavior (i.e., the execution tree of the Web Service) is represented by a (deterministic) Finite State Machines (FSMs). It is shown that a composition for an external schema represented as a FSM is constituted by a Mealy FSM (MSFM). Then synthesizing such a MFSM is achieved by reducing the problem of composition existence into satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL).

2.3 AI Planning

2.3.1 Classical Planning

Most of the planning approaches rely on a general model, the model of state-transition systems. In a state-transition system there are finite or recursively enumerable set of states, actions and events along with a transition function that maps a state, action, event tuple to a set of states. Given a state transition system, the purpose of planning is to find which actions to apply to which states in order to achieve some objective, starting from some given situation.

Classical planning is mainly based on the initial modeling of the STRIPS [16] system. In this representation a state is represented by a set of ground literals expressed in a first-order language. An action is an expression specifying which first-order literals 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.

This representation is insufficiently expressive for some real domains. As a result, many language variants have been developed. Action Description Language (ADL) [36] is an important variation. ADL extends STRIPS representation by explicitly including negative literals in the state, having conditional effects for operators and allowing existential variables and disjunctions in goal formulas. Penberthy and Weld [39] developed a partial order planning algorithm named UCPOP [40] to handle a significant subset of ADL action representation.

2.3.2 HTN Planning

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 or how prior decompositions occurred. Such hierarchical modeling is the core of the OWL-S [34] process model to the point where the OWL-S process model constructs can be directly mapped to HTN methods and operators as shown in [47].

SHOP2 [33] 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 planning 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 during planning.

2.3.3 Planning with Incomplete Information

The XII [20] is a general-purpose planner which was originally designed to help an autonomous agent plan in the presence of incomplete information. Other planners of this genre include Cassandra [11] and IPEM [2]. XII can handle both causative goals and knowledge-information goal. As an example one could use XII to first compress all the ps files in a directory and then list all files which are below a certain size. The first is a causative goal, while the second is an information-gathering goal, whose outcome might change based on the causative actions that the agent might take before considering this goal. In this case, some of the postscript files which were above the size threshold before the compression was done, my get below the threshold after the compression, and thus become eligible tuples for the information gathering goal.

XII can in principle be used to solve the pure information gathering problems, with source calls modeled as information gathering actions with knowledge effects. However, use of XII for pure information gathering turns out to be an over-kill. This is because the absence of causative changes to the environment around the information gathering agent (the contents of the information sources are not modified by the queries sent to them) vastly simplifies the planning problem, facilitating specialized methods such as the ones described in [27]. However, XII methodology may be useful once we consider variants of the information gathering problem that model updates to sources (either made by the information gatherer, or more likely, by the source providers).

PUCCINI [20] is an extension of XII but has a richer language to specify actions and goals and handles verification links. Interleaving planning with execution builds on the approach used in IPEM. Unlike IPEM, PUCCINI can represent information goals as distinct from satisfaction goals.

Knoblock et. al [3] developed the Sage system which is originally intended to be a query planner for the SIMS project, that deals with heterogeneous distributed databases. Sage assumes information source descriptions are complete, and that no source has query constraints, Sage casts the information gathering problem as a query reformulation problem. Sage uses a modified version of UCPOP to search for the correct sequence of reformulation operations that will transform the user's query into an equivalent query only on information sources.

3 Preliminary Work

In this section, I describe my preliminary work on Web Service composition. Section 3.1 explains the interactive composition of Web Services where a user builds a composition with the help of a semi-automated tool I built. The tool uses Web Ontologies to find and filter Web Service matches. Issues such as the generation of OWL-S descriptions from WSDL specifications and use of concept-mapping Web Services to improve multi-ontology matches is also described in this section. Section 3.2 describes how to automate the composition of Web Services using HTN planning. The section explains how the OWL-S processes was mapped to HTN task descriptions and includes the proof for soundness and completeness of the plans generated after this mapping. Section 3.3 describes my initial work on information gathering during planning and presents some of the preliminary results obtained. Section 3.4 describes the issues related to using Web Ontologies to describe preconditions and effects of Web Services. The integration of a Semantic Web reasoner with an HTN planner is examined and the problems caused by the extra expressivity of Web Ontologies and their distributed nature are discussed. Lastly, section 3.5 describes my work on implementing a Semantic Web reasoner and how this relates to the various reasoning tasks that were used in different parts of my preliminary work .

3.1 Interactive Composition of Web Services

As a starting point, I have developed an interactive tool to partially automate the Web Service composition process. The composition of Web Services is achieved in a goal-directed fashion where the composition is gradually generated with a forward or backward chaining of services. At each step, a new service is added to the composition and further possibilities are filtered based on the current context and user decisions.

Building the composition step-by-step is very intuitive for many cases. For example, consider the task of making the necessary travel arrangements for a trip. The first step is to book a means of transportation. You start by finding the services that let you make reservations for transportation. Then you need to filter these services because not all of the services are relevant to your current task—e.g. ones that does not provide transportation to your destination or ones that have no availability at the desired dates should not need to be considered. Filtering may be further used to help determine the service that best fits for your personal preferences, such as accepting a certain credit card or serving particular destinations with non-stop flights. After this step is resolved, you can continue the composition process by finding compatible services. Perhaps you have a clear idea of what further tasks you'd like to accomplish with this composition or perhaps just seeing the available, compatible services will suggest further goals. Just as with business or consumer services, we expect propinquity to be a key factor in de-

Making Travel Arrangements
1. Book transportation
1.1. Find transportation services
1.2. Filter out the services which has no availability at the desired dates
1.3. Select a service that accepts your credit card, offers a good price, etc.
2. Make hotel reservation (feed date of arrival information from previous service to this one)
...
3. Record expenses in your financial organizer (compute total of expenses from previous steps)

Table 1: A step by step composition of a service that will make the travel arrangements for a trip

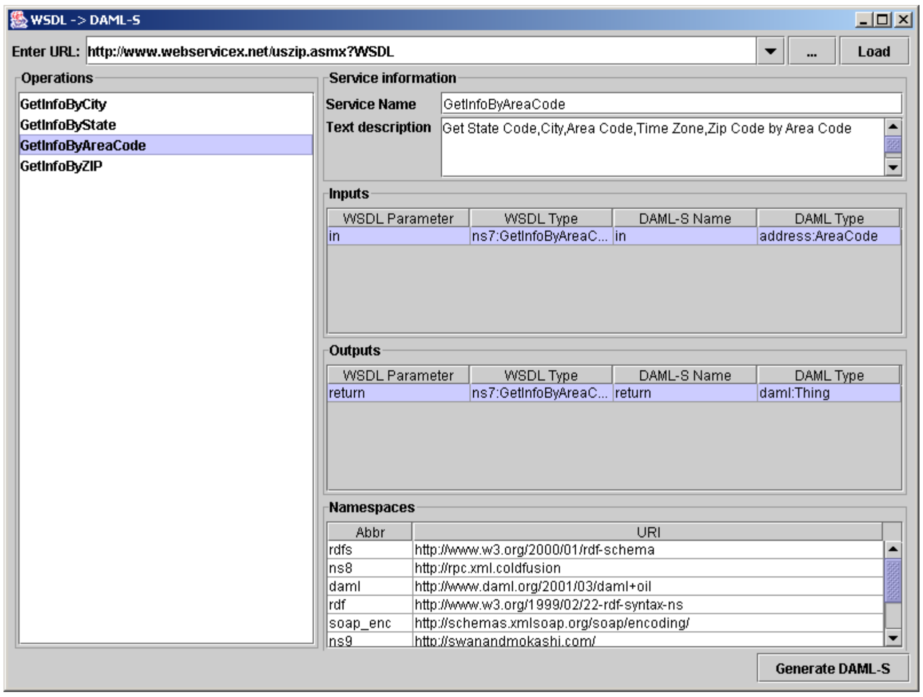
termining desirable compositions, particularly when the "extra" services are not strict requirements of the current task.

3.1.1 Creating Semantic Service Descriptions

Partial automation of composition can effectively be done when the Web Services have fairly rich descriptions that will help to find the relevant Web Service matches. As discussed in section 2.2.2, using Semantic Web ontologies to describe Web Services provides possibilities to automatically generate flexible matches. Unfortunately, it is not possible to find a large number of Web Services described in OWL-S. On the hander hand, there is an increasing number of WSDL-described web services available on the Web, both from independent developers and large companies (e.g., Amazon and Google). Annotating these web services with OWL-S provides a good opportunity for us to access a lot of semantically described, executable services.

For this reason, I worked on to partially automate the derivation of OWL-S descriptions from WSDL descriptions. For each *operation* a WSDL document describes, the document will provide a description of the input and output messages and their substructure for that operation. Normally we take a WSDL operation to correspond to an OWL-S AtomicProcess, with the parameters of that process corresponding to various message parts. In nearly all WSDL documents, the content of message parts are described by XML Schema datatypes, quite often complex types (that is, types which describe elements with possible attribute or subelement structure). Since parameter type compatibility is a critical part of the interactive composition method, it is very important that the service description supplies sufficiently expressive types.

For many purposes it is preferable to have the parameter types of OWL-S services be OWL classes, as it would allow for more flexible matching and more natural OWL-based descriptions. Since we are already augmenting the information in a WSDL description, it seems reasonable to do so with the types as well. Thus, we treat the WSDL supplied types as descriptions of the "wire format" of the service parameters, that is, the serialization of the values actually used by our process. We extended the OWL-S


WSDL -> DAML-S

Enter URL: http://www.webservicex.net/uszip.asmx?WSDL

Operations

- GetInfoByCity
- GetInfoByState
- GetInfoByAreaCode
- GetInfoByZIP

Service information

Service Name: GetInfoByAreaCode

Text description: Get State Code,City,Area Code,Time Zone,Zip Code by Area Code

Inputs

WSDL Parameter	WSDL Type	DAML-S Name	DAML Type
in	ns7:GetInfoByAreaC...	in	address:AreaCode

Outputs

WSDL Parameter	WSDL Type	DAML-S Name	DAML Type
return	ns7:GetInfoByAreaC...	return	daml:Thing

Namespaces

Abbr	URI
rdfs	http://www.w3.org/2000/01/rdf-schema
ns8	http://rpc.xml.coldfusion
daml	http://www.daml.org/2001/03/daml+oil
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
soap_enc	http://schemas.xmlsoap.org/soap/encoding/
ns9	http://swanandmokashi.com/

Figure 1: A tool to automate translation from WSDL descriptions to OWL-S

Grounding to all for the inclusion of marshaling and unmarshaling functions which our OWL-S executor can use to coerce XML Schema values to OWL individuals and back.[2] These functions are, by default, encoded as XSLT stylesheets. For example, an unmarshaling function is written as an XSLT transformation from XML fragments matching the specific XML Schema type to an RDF graph serialized in the RDF/XML exchange syntax. That graph encodes the relevant assertions about the individual which is the actual input to the service. Marshalling functions are implemented as the inverse transformation. Using published XSLT obviates the need for the OWL-S executor to be extended with specific type coercion functions — it just needs a generic XSLT processor, perhaps running as a remote service. The downside is that due to the extremely free syntax of RDF/XML (especially, the plurality of equivalent forms), it is difficult to write XSLT that can handle all the legal serializations of a given RDF graph, and the resulting stylesheet is difficult to understand and maintain.

Clearly, writing such transformation functions by hand is not feasible. Marshalling and unmarshaling functions already can be a source of subtle bugs as they require a deep understanding of both source and target formalism, a good understanding of the

[2]These extensions, with further development by the OWL-S coalition, were subsequently included in OWL-S. These extensions and their implementation were done in collaboration with Fujitsu Labs of America, College Park, with extensive feedback from Ryusuke Masuoka.

services both on the WSDL side (i.e., of the operational semantics of the service) and on the OWL-S side (i.e., of how the descriptions affect the various of OWL-S related inferences). Adding essentially irrelevant and idiosyncratic details of a specific linear syntax for RDF compounds the problem. Unfortunately, current standard solutions tend to compromise interoperability. In our system, since we control all our execution engines (in fact, we reuse a single implementation), we can require a specific profile of RDF/XML that avoids confusing or redundant constructs. Clearly if other engines do not generate that profile, then our XSLT transformations can fail. Also it is unclear that, even with a suitably designed profile, the necessary XPath queries will be sufficiently obvious and transparent to the programmer. Finally, while feeding the XSLT processor some XML allows for great flexibility, both in choice of implementation of processor and of the specific instance of some processor, it is unlikely that the internal representation of the individual will be, say, W3C DOM trees, so there is the constant need for additional data conversion.

All three issues would be dealt with by the incorporation of an RDF and OWL sensitive query language (such as RDQL or Versa) into the XSLT, or perhaps XQuery, standards. Even if generic XSLT or XQuery processors generally failed to include such extension, it would provide a standard and appealing target for OWL-S engines to implement; and, even if the query languages were not ideal, they would have both less of a conceptual gap and less of an implementation gap than XPath queries.

An appealing alternative to either technique is to use a higher level mapping language, perhaps along the lines of MDL [46] as proposed in [37]. If the mappings could be compiled to XSLT or other transformation languages, there would be an enormous gain in portability, and by eschewing the general expressive power of programming languages like XSLT, there might be a significant gain in transparency and analyzability. Unfortunately, the design of such a language covering the entire expressivity of OWL is a formidable task.

3.1.2 Using Web Ontologies for Partial Automation of Composition

I have built a system to provide support for our interactive composition approach using semantic service descriptions. Filtering and selection of services is achieved by using matchmaking algorithms similar to those implemented in [35], [21] and [29]. extended this algorithm to consider the subsumption relation between the request and advertisement profiles considered as whole concepts.

Our system uses the same basic typology of subsumption based matches, but in some contexts we match based on the subsumption of the entire profiles, and in other contexts we use subsumption only to directly match individual parameters.

The system has two separate components. An inference engine is responsible for storing service advertisements and processing match requests. The inference engine is Pellet [38], the OWL-DL reasoner I implemented. The other component of the system is the composer where the workflow of service composition is generated. The composer communicates with the inference engine to discover possible matches and present them to the user. It also lets users to invoke the completed composition on specific inputs.

The composer lets the user create a workflow of services by presenting the possible choices at each step. The user is first presented with all the available services registered

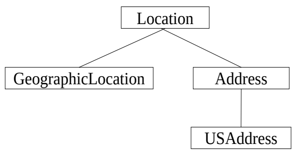


Figure 2: A simple hierarchy of location related concepts

to the system. This first step is totally unguided. Each subsequent step of the composition makes use of two sorts of matching, on IOPEs (which is fully automated) and on other service parameters. Forms for entering constraints on the service parameters are generated from the ontologies defining those parameters. In any step, the final selection of the specific service is done by the user.

3.1.3 Matching on IOPEs

At each step of the composition, a list shows the IOPE compatible services that can be added to the composition. When a service is selected from the list, the composer presents as options those services whose output could be fed to the current service as an input. Suppose the selected service accepts an input of type *Address* which is defined in a certain ontology where the concept hierarchy is shown in Figure 2. We would like to find the services which have an output that is compatible with this type. An output of a service would be considered compatible if it was of type *Address* or another concept which is subsumed by *Address*, i.e. *USAddress*. When the output of a service is subsumed by the input, the output type can be viewed as a specialized version of the input type and these services can still be chained together. However, a service whose output is *Location* could not be composed with this service since *Address* concept will most likely have additional properties and restrictions on the existing properties of *Location*.

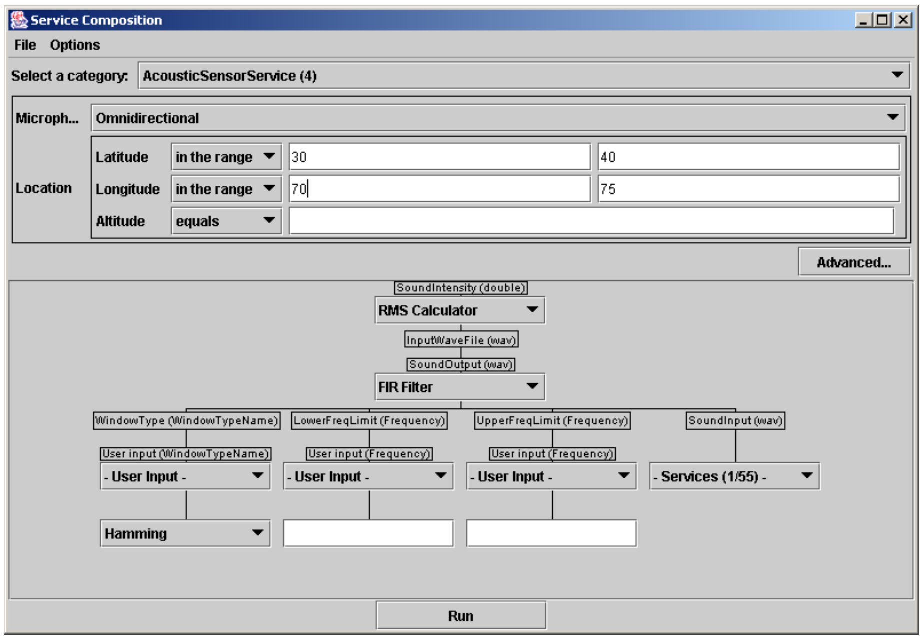
Clearly, only Exact and PlugIn matches between the parameters of ServiceProfiles would yield useful results at this step. For service selection, we need match on individual parameters types instead of whole profiles, as we consider all type compatible services to be reasonable “next steps” of a composition. One interesting extension would be to consider certain service parameters against global constraints as part of service compatibility. For example, suppose before starting the composition process, the user enters an overall price limit on the composition. At any step, the system sums the values of all cost service parameters of the currently composed services, and uses the difference between that sum and the set limit to filter potential next steps.

The ordering of the result displayed in the list is based on the degree of the match. The Exact matches are more likely to be preferred in the composition and these services are displayed at the top of the list. The PlugIn matches are presented after the Exact matches and PlugIn matches are ordered according to the distance between the two types in the ontology tree.

3.1.4 Filtering on Service Parameters

The number of services displayed in the list as possible matches can be extremely large. For example, a power grid or telephone network might have many thousands of sensors each providing several services. This will make it infeasible for someone to scroll through a list and choose one of the services simply by name. Furthermore, even if the number of services is low, the service names themselves may not be contain enough information to let a user know what they do. When the name of the service does not help to distinguish the services, we turn to the other service parameters, such as location, to help determine the most relevant service for the current task. Thus, a sensor description, linked to a particular service, can be queried as to the sensor's location, type, deployment date, sensitivity, etc.

The ServiceProfile hierarchies defines a classification which is used at the first level of filtering. By selecting a profile category from the list, user limits the shown available choices whose ServiceProfile matches with the selection. We examine the definitions of the various ServiceProfiles to build various user input forms for specifying further constraints on the desirable services.


Select a category: AcousticSensorService (4)

Microph... Omnidirectional

Location
Latitude in the range 30 40
Longitude in the range 70 75
Altitude equals

SoundIntensity (double)
RMS Calculator
InputWaveFile (wav)
SoundOutput (wav)
FIR Filter
WindowType (WindowTypeName) — User input (WindowTypeName) — - User Input - — Hamming
LowerFreqLimit (Frequency) — User input (Frequency) — - User Input -
UpperFreqLimit (Frequency) — User input (Frequency) — - User Input -
SoundInput (wav) — - Services (1/55) -

Figure 3: Filtering is used to see only omnidirectional acoustic sensors that are located at a latitude between 30-40 and a longitude between 70-75. It is seen that only one of 55 services satisfy these constraints

Consider a example in the sensor network where we want to select a specific sensor service. With no other restriction, the system will present every available sensor service. This is better than presenting all the services, but the remaining choices can still be overwhelming. If the user chooses to filter the results to the services with *Acoustic-SensorServiceProfiles*, that decreases the number of matches significantly. The composer then queries the inference engine about the possible service parameters of the selected service type. Based on the answer returned from the engine, the composer creates a GUI panel in which the user can enter constraints for the properties of the services as shown in Figure 3.

The user's constraints are combined in a service request profile. The service request is sent to the inference engine and the result of this new query is applied to the previous result set. The services that do not satisfy the current constraints are removed from consideration. The matchmaking for this step can use Relaxed matches as well as Exact and PlugIn matches. Using Relaxed matches will probably increase the choices presented allowing the user to make a more flexible selection. Relaxed matches are permissible because we already know that the set of services the user is considering are compatible in this context.

3.1.5 Improving IOPE Matching with Ontology Translation Services

With both IOPE matching and service parameter filtering there is a strong need for a suitable set of service descriptions of sufficient and compatible detail to support, for IOPE matching, the appropriate subsumptions and, for service parameter filtering, intelligible form based queries. It is straightforward to elaborate the service parameter filter forms by extending the definitions of the concepts used to describe those parameters. We expect that such extension will be done using standard ontology editing tools.

We have already discussed improving IOPE matching by converting the IO type descriptions from XML Schema datatypes to OWL classes. In that process, the choice of target OWL class is critical to generating matchmaking hits. The Semantic Web is likely to have a large number of somewhat overlapping ontologies, that is, ontologies which have fairly similar, but distinct concepts. If service description authors choose different, but relevantly equivalent, classes to unmarshall their XML Schema datatypes to, the system will fail to match intuitively compatible services. Ideally, some sort of concept or ontology mapping would make these relevant equivalences transparent to the system. Aside from the normal OWL-DL constructs for equating classes, we have the concept of a *TranslatorServiceProfile*, that is, of services whose entire job is to take the description of an OWL individual against one ontology, and produce the relevantly equivalent set of assertions against another.

However, there is an important sense in which these services are unimportant to the composition process. Rather, they are *only* important insofar as they promote the composition of other services which actually move the user closer to her goal. They are not suggestive of interesting further steps, thus are merely a burden on the user. To eliminate this, we do not actually present the translation services to the user, but rather created "fused" services on the fly. A fused service is a chain of translation services terminating in a non-translation service. The fused service is presented as a type compatible non-translation service, thus increasing the number of substantial options at any particular step. Details about the mapping can be found in [42].

3.2 Automated Composition of Web Services Using HTN Planning

Web Service descriptions can be extended to include information such as preconditions and effects. OWL-S description language uses these constructs give more information about what the service does. It is possible to map such descriptions to planning operators and 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 that achieves the objective.

We believe that HTN planning is especially promising for this purpose, because the concept of compound tasks in HTN planning is very similar to the concept of composite process descriptions. A Web Service workflow that has a complex structure with many different execution paths can be modeled as an HTN method. This information can be fed to a HTN planner as a planning domain and planner would compose a sequence of atomic processes that would consitute a valid deccomposition of the original composite service.

There are several ways in which 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. Also HTN planning scales well to large numbers of methods and operators as method decompositions provide means to prune the search space by ignoring unrelated method descriptions.

In the following sections first encoding OWL-S process models as SHOP2 domains is explained, then definition of how to formalize a Web Service composition problem as SHOP2 domain is shown. Then the soundness and correctness of the plans generated by SHOP2 is proven with respect to the situation-calculus semantics of OWL-S given in [31] and [31].

3.2.1 Encoding OWL-S Process Models as SHOP2 Domains

In [47] we have provided the details of a mapping algorithm that translates the OWL-S process descriptions to SHOP2 planning domains. The encoding of Web Service descriptions to HTN domains is achieved as follows:

- Each atomic process with effects is encoded as a SHOP2 operator that simulates the effects of the world-altering Web Service.
- Each atomic process with output is encoded as a SHOP2 operator[3] whose precondition include a call to the information-providing Web Service.
- Each simple or composite process is encoded as one or more SHOP2 methods.

[3]These processes are encoded as "book-keeping" operators so they do not appear in the final plan.

These methods will tell how to decompose an HTN task that represents the simple or composite process.

This mapping assumes that all atomic processes defined in OWL-S process model can either have effects or outputs, but not both. 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 don't 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.

It is also assumed that there is no OWL-S composite process in the input that uses *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*. The details of the encoding can be found in [47].

3.2.2 Encoding OWL-S Web Service Composition Problem as SHOP2 Planning Problem

Narayanan and McIlraith [32] give a formal semantics for OWL-S in terms of the situation calculus [41] and Golog [28]. The situation calculus in a first-order language for reasoning about action and change. In the situation calculus, the state of the world is described by 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 (Figure 4) 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 $\vec{a}$, such that $D \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that Golog program δ starting at S_0 will legally terminate in situation $do(\vec{a}, S_0))$ where $do(\vec{a}, S_0))$ is used to abbreviate the following expression $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0))$. Thus, $a_1, \ldots, a_n$ are the actions that realize Golog program δ, starting in the initial situation, S_0.

The semantics given in [32] and [31] 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 3.1 (OWL-S Service Composition) *Let $K = \{K_1, K_2, \ldots, K_m\}$ be a collection of OWL-S process models satisfying the assumptions listed in Section 3.2.1,*

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

Figure 4: A subset of Golog constructs to create complex actions that are relevant to OWL-S constructs.

C be a possibly composite process defined in K, S_0 be the initial state, and $P = (p_1, p_2, \ldots, p_n)$ be a sequence of atomic processes defined in K. Then P is a composition for C with respect to K in S_0 iff 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 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 [31]) that need 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 (Figure 5) 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) [19] 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 [28] 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.

```
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 $o$ = ($h$ $Pre$ $Add$ $Del$) in $D$ such that
                 $h$ unifies with $t$ and $s$ satisfies $Pre$
6            if no such $o$ exists then return failure
7            Let $s'$ be $s$ after deleting $Del$ and adding $Add$
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 $m$ = ($h$ $Pre_1$ $T_1$ $Pre_2$ $T_2$ $\ldots$) in $D$ such that
                 $h$ unifies with $t$
12           Find the task list $T_i$ such that
                 $s$ satisfies $Pre_i$ and does not satisfy $Pre_k$, $k < i$
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
```

Figure 5: A simplified version of the SHOP2 planning procedure.

Theorem 3.2 *Let $K = \{K_1, K_2, \ldots, K_m\}$ be a collection of OWL-S process models satisfying the assumptions listed in Section 3.2.1, C be a possibly composite process defined in K, S_0 be the initial state, and $P = (p_1, p_2, \ldots, p_n)$ be a sequence of atomic processes defined in K. Then P is a composition for C with respect to K in S_0 iff P is a plan for planning problem (S_0, M_C, D) where M_C is the SHOP translation for process C and D is the SHOP domain created from K.*

Proof 3.3 *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, $f(\vec{x})$ 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., $f(\vec{x}, s)$. 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. $f(\vec{x})$ is true in the initial state of SHOP2 iff $f(\vec{x}, S_0)$ 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 $f(\vec{x})$ after applying the effects of an operator will be equal to the truth value of $f(\vec{x}, do(a, s))$ 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:*

Hypothesis *For a given OWL-S process C, P is a plan for the planning problem (S_0, M_C, D) iff $\Sigma \models Do(\delta_C, S_0, do(\vec{a}, S_0)))$ where $\vec{a} = [a_1, a_2, \ldots]$ 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 $\vec{a} = [a]$. 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 two*[4] *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 δ_C = $\delta_{C_1} \mid \delta_{C_2}$ 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 $\vec{a}$ 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 $\vec{a}$ 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*

[4]The 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

= (C $\emptyset$ (M_{C_1} M_{C_2})). The corresponding Golog program for C is δ_C = δ_{C_1} ; δ_{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 $\vec{a_1}$. If the action sequence recorded between situations s^* and s' is $\vec{a}_2$ then the final situation s' represents the concatenated sequence $\vec{a}$ =[$\vec{a}_1$, $\vec{a}_2$]. Calling SHOP2(s, M_{C_1}, D) will return P_{C_1} and from our hypothesis we know that it is equivalent to the action sequence $\vec{a_1}$. 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 $\vec{a_1}$. The 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 $\vec{a}$.

If-Then-Else *Suppose C is a composite OWL-S process defined with an* 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 $cond$ M_{C_1} $\emptyset$ M_{C_2}). Corresponding Golog program for C is δ_C = (**if** $cond$ **then** δ_{C_1} **else** δ_{C_2} **endIf**) and the semantics is defined as*

*Do(**if** $cond$ **then** δ_{C_1} **else** δ_{C_2} **endIf**, s, s')*
= Do(($cond$?; δ_{C_1}), s, s') $\vee$ Do(($\neg cond$?; δ_{C_2}), s, s')
= (cond[s] $\wedge$ Do(δ_{C_1}, s, s')) $\vee$ ($\neg$cond[s] $\wedge$ Do(δ_{C_2}, s, s'))

The expression $cond[s]$ evaluates to true whenever the fluent $cond$ is true in situation s. Suppose $\vec{a}_1$ is the action sequence for the situation δ_{C_1} and $\vec{a}_2$ is the action sequence for the situation δ_{C_2}. If s satisfies $cond$ then the result for δ_C will be $\vec{a}_1$ otherwise result will be $\vec{a}_2$. From our hypothesis we know for any possible $\vec{a}_1$ (or $\vec{a}_2$) 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 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 translation to be M_C = (C $cond$ (C_1 C) $\emptyset$ $\emptyset$). Corresponding Golog program for C is δ_C = (**while** $cond$ **do** δ_{C_1} **endWhile**) and the semantics is defined as*

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

*This definition includes the nondeterministic iteration operation * which has a second-order semantics [28]. We will use the restricted version of Golog as defined in [31] where the 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($\neg cond$?, 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 $\vec{a}$ is a valid action sequence for δ_{C_1}. From our hypothesis we know for each action sequence $\vec{a}$ 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([$cond$?; $(\delta_{C_1})^1$; $\ldots$; $cond$?; $(\delta_{C_1})^k$; $\neg cond$?], s, s') hence the action sequence will be [$\vec{a}_1$, ..., $\vec{a}_k$]. 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 $s, s_1, \ldots, s_{k-1}$ 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 kth 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, ..., 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 [31]. 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 doesn't 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 [31]. Intuitively, IRP assumption says that

- Information-providing services should be executable in the initial state, and
- 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 3.2.3) 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 don't allow the information-providing services 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 encoding of information-providing services to use standard operators rather than "book-keeping" operators.

3.2.3 Implementation

To realize these 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 3.2.1, 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. 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.

The system was tested on a domain which we created based on the scenario described in the Scientific American article about the Semantic Web [6]. This scenario describes two people who are trying to do arrangements for their mother's medical needs. They need to fill the prescription given by the doctor at a pharmacy, make appointments for two different treatments, and make an appointment with the doctor for a follow-up meeting. The planning problem is to come up with a sequence of appointments that will fit in to everyone's schedules, and at the same time, to satisfy everybody's preferences, i.e. time and distance constraints.

We ran this domain on our 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 the CVS Online Pharmacy Store. It was not possible to use real services for some of the services either because they were not 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 access to the 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 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.

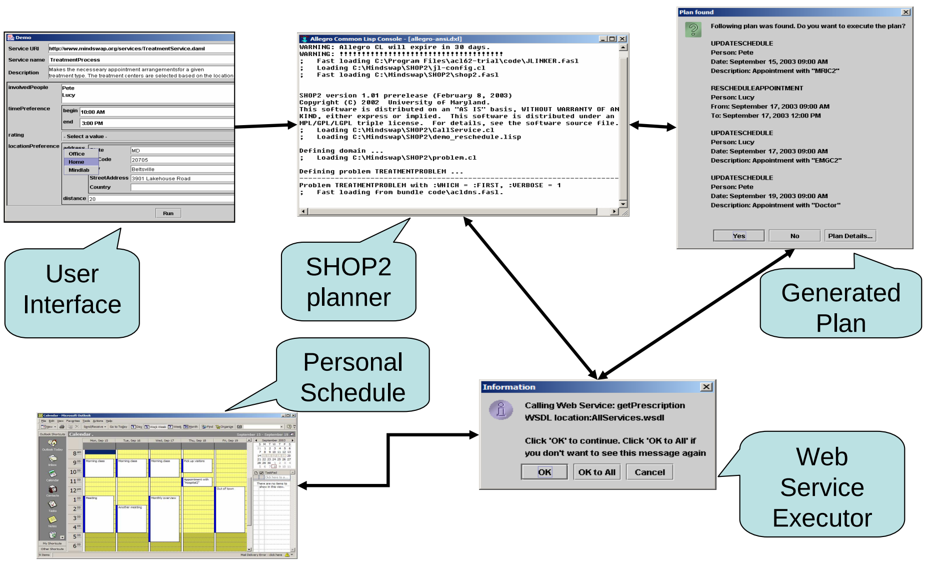
- We also encoded a description of how to compose Web Services for the tasks described in this example scenario. The description is given as a OWL-S composite process that is composed of several other composite processes that are defined as sequence, choice or unordered processes. This OWL-S description constitutes the top level composite process described in Section 3.2.1 and is translated into a SHOP2 domain for planning. We encode most of the constraints mentioned above as preconditions of Web Services. 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.

Figure 6 shows the various components of the system [5] 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 an "abstract" task list, that is, a set of tasks which can be achieved in a variety of ways. In order to "execute" this service we must decompose these abstract tasks into actions (services) that we can actually invoked. 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 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

[5]This system was demonstrated in the Developer's Day of the 12th WWW conference in Budapest, Hungary



Service URI: http://www.mindswap.org/services/TreatmentService.daml
Service name: TreatmentProcess
Description: Makes the necesseary appointment arrangementsfor a given treatment type. The treatment centers are selected based on the location
involvedPeople: Pete, Lucy
timePreference: begin 10:00 AM, end 3:00 PM
rating: - Select a value -
locationPreference: State MD, Code 20705, Beltsville, StreetAddress 3901 Lakehouse Road, Country; distance 20

```
Allegro Common Lisp Console - [allegro-ansi.dxl]
WARNING: Allegro CL will expire in 30 days.
WARNING: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;   Fast loading C:\Program Files\acl62-trial\code\JLINKER.fasl
;   Loading C:\Mindswap\SHOP2\jl-config.cl
;   Fast loading C:\Mindswap\SHOP2\shop2.fasl


SHOP2 version 1.01 prerelease (February 8, 2003)
Copyright (C) 2002  University of Maryland.
This software is distributed on an "AS IS" basis, WITHOUT WARRANTY OF AN
KIND, either express or implied.  This software is distributed under an
MPL/GPL/LGPL triple license.  For details, see the software source file.
;   Loading C:\Mindswap\SHOP2\CallService.cl
;   Loading C:\Mindswap\SHOP2\demo_reschedule.lisp

Defining domain ...
;   Loading C:\Mindswap\SHOP2\problem.cl

Defining problem TREATMENTPROBLEM ...
------------------------------------------------------------------------
Problem TREATMENTPROBLEM with :WHICH = :FIRST, :VERBOSE = 1
;   Fast loading from bundle code\acldns.fasl.
```

Plan found

Following plan was found. Do you want to execute the plan?

UPDATESCHEDULE
Person: Pete
Date: September 15, 2003 09:00 AM
Description: Appointment with "MRIC2"

RESCHEDULEAPPOINTMENT
Person: Lucy
From: September 17, 2003 09:00 AM
To: September 17, 2003 12:00 PM

UPDATESCHEDULE
Person: Lucy
Date: September 17, 2003 09:00 AM
Description: Appointment with "EMGC2"

UPDATESCHEDULE
Person: Pete
Date: September 19, 2003 09:00 AM
Description: Appointment with "Doctor"

Information

Calling Web Service: getPrescription
WSDL location:AllServices.wsdl

Click 'OK' to continue. Click 'OK to All' if you don't want to see this message again

Figure 6: A snapshot of the running system and the interaction between different components of the system

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.

3.3 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 during planning. Contrariwise, at composition execution time, the primary interest is in the execution of world-altering processes. Indeed, in the implementation 3.2.3 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 unaesthetic. 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.

This approach successfully gathers information during planning time but still lacks the flexibility of a general-purpose solution because it relies on the fact that information-providing services are hard-coded in the given domain information. However, in a more realistic situation the domain would not include such descriptions, i.e. the services that needs to be executed to gather the information. It is required that planner itself figures out when and how to gather the information.

In [26] we have relaxed this restriction such that the information providing services do not need to be explicitly specified in the initial description. An arbitrary query mechanism can be used to select the appropriate Web Service on the fly when the information is needed. We have developed the ENQUIRER system which extends SHOP2 by gathering information during planning as needed.

Executing Web Services to get the information will typically take longer time than the planner would spend to generate plans. In some cases, it will not be known a priori which Web Service gives the necessary information and it may not be possible at all to find those services. Furthermore, in some cases the found service cannot be executed because the service requires some password that the user cannot provide or the service is inaccessible due to some network failure. ENQUIRER was designed to tackle this problem can continue planning while the information-providing services are still running.

3.4 Using Ontologies During Planning

OWL-S descriptions mainly use Semantic Web ontologies to specify input and output types. All existing versions of OWL-S have left the particular language for encoding preconditions and effects unspecified. Consequently, the mapping algorithm in section 3.2.1 assumed that the expressions were written in SHOP2's encoding. However, these conditions should (and as the forthcoming OWL-S 1.1 version forces) and will be also be written in OWL. In order to evaluate these precondition formulas written in OWL, planners must understand the semantics of OWL. Unfortunately, the typical logic for expressing preconditions and effects in a planning system is quite differently expressive than RDF and OWL do. Therefore, planning against the sorts of encodings of the world state that is expected to exist on the Semantic Web will be different than the planners can handle.

I have worked on integration of a Semantic Web reasoner with SHOP2 planner in order to overcome this problem. The integration means that all of the planner's interaction with the state, i.e. querying and updating, will be done by the reasoner. And most important of all the world state itself is actually represented as an OWL knowledge base. Evaluation of preconditions is done by the reasoner and any statement entailed by the KB is assumed to be true in the state.

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 generates sound and complete plans as long as its theorem proving is sound and complete.

3.4.1 Operator Definitions

We want to change the classical planning operator definitions such that preconditions and effects will be written in 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 has been used in planning community and how these can be used in our context.

The original STRIPS [16] language allowed to use arbitrary well-formed formulas in first-order logic for preconditions and effects. However, defining a semantics for this formulation was problematic [30]. 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 treats 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 person is registered may mean that we don't know if person is registered or not. 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) [18] 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 don't 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

```
(: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)))
```

Figure 7: A simplified service description where person $?p$ makes an appointment with doctor $?d$ at time $?t$.

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 does 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 [33].

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 [17]) 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 7 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.

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 3.4.3. 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 and we will discuss this issue in more detail in section 3.4.3.

3.4.2 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 [24]. 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 precondition 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 don't know who that child is. On the other hand, when we want to bind the variable ?c to an existing individual then 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. OWL query language proposal [15] suggests to label the variables as *must-bind*, *may-bind*, and *dont-bind* to control this behavior. This is also consistent with the 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 and a method may use it as an input of a subtask. An *active* variable should always be bound to an actual individual to ensure that we always have ground terms. *Inactive* variables can be labeled as *dont-bind* or *must-bind* according to the 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 2.3.1, current state of the art planning systems use

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

Figure 8: A simple book buying service saying that the available limit on the credit card should be higher than the price of the book

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 [24].

Universally quantified expressions in preconditions also creates a problem with the open world semantics. Consider the following 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 quantified 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 a daughter (again we just don't know who she is).

In most real world problems preconditions involve some kind of numerical computation (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 3.4.2. 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 minValue facet. 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 3.4.2, so it may be preferable to have another module that processes these expressions.

3.4.3 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 in the KB. 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 the planner 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 in the KB.

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 3.4.1).

Unfortunately, restricting the usage of derived predicates in effects makes it nearly impossible to model any action in OWL. The following table summarize the conditions that causes an OWL property p to be a derived predicate:

When p is a derived predicate	**In DL syntax**	**How it is derived**
p has a subproperty	$q \sqsubseteq p$	$q(x, y) \rightarrow p(x, y)$
p has an equivalent property	$p = q$	$q(x, y) \rightarrow p(x, y)$
p has an inverse property	$p = q^{-}$	$q(x, y) \rightarrow p(y, x)$
p is a symmetric property	$p = p^{-}$	$p(x, y) \rightarrow p(y, x)$
p is a transitive property	Transitive(p)	$p(x, y) \wedge p(y, z) \rightarrow p(x, z)$

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 the class for a class C meets any of the following conditions:

When C is a derived predicate	**In DL syntax**	**How it is derived**
C has a subclass	$D \sqsubseteq C$	$D(x) \rightarrow C(x)$
C has an equivalent class	$C = D$	$D(x) \rightarrow C(x)$
C is defined to be the range of a property	range(p, C)	$p(y,x) \rightarrow C(y)$
C is defined to be the domain of a property	domain(p, C)	$p(x,y) \rightarrow C(x)$

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 his/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.

3.5 Reasoning with Semantic Web Ontologies

Ontologies play an important role in describing Web Services. Interpreting the information in these ontologies becomes a crucial task for understanding Web Service capabilities and their behavior.

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 3.4.2, 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 than we have to generate answer sets for these variables. The existing conjunctive ABox query answering algorithms [24, 25] 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. [23]. For example, the query (A rdf:type $Rover$) is entailed by the KB S if and only if $\{S \sqcup (A$ rdf:type complementOf($Rover$))$\}$ 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:∃.manufacturedBy $\{L\} \sqcap$ ∃hasCPU. ($\{CPU\} \sqcap$ ∃cpuType.$\{Centrino\}$)). The query can now be answered by adding the negation of this concept to individual A and then check 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 replacing 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. [25] 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 substitute 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 try every combination of bindings where we do a total of $33^4 \approx 1186000$ consistency tests. The optimization described above would help us to reduce the number of candidates so we wouldn't 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 [22]. 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 if we combine this structural inspection with optimized retrieval we can get complete and fast results. 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 instance of the concept $\exists p.\{o\}$ to get the remaining answers. Note that, if all the individuals are related with explicit assertions then only one consistency check 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 these trivial bindings.

4 Future Directions

This section describes the challenges that I have identified as a result of my preliminary work, and discuss how I intend to address those challenges as my future work.

4.1 Planning with Web Service Descriptions

My preliminary work for using planning for Web Service composition was based on the assumption that Web Services are described in a fairly rich and essentially planning-oriented description language such as OWL-S. The process ontology of OWL-S is designed to describe Web Services similar to planning operators. Web Services have explicit precondition and effect descriptions and composite services may be modeled similar to compound HTN tasks.

Describing preconditions and effects of Web Services using Web Ontologies introduces some challenges. Handling the expressivity of Web Ontologies during planning is non trivial. For example, as discussed in section 3.4.3, using derived predicates in effect descriptions is prohibited in planning for the sake of soundness. However, the expressivity of OWL causes almost all practical Web Service descriptions violate this restriction. It is even harder to ensure this condition on the Web where anyone can extend an existing ontology causing a Web Service description violate the restriction. I will investigate if and how these expressive descriptions can be handled in planning. It might be possible to extend planner's inferencing capabilities to handle this expressivity but it might also be required to find some alternative ways of writing these descriptions.

It is also important to note that not all Web Services are described in a planning-oriented language. Most of the Web Service descriptions that are publicly available on the Internet do not have explicit precondition/effect specification. These services are merely described in terms of their functional signature, i.e input and output types. In general, there is a tendency to describe Web Services using the structure of messages and the message exchange patterns between Web Services. Another commonly used method is to use taxonomies, such as UNSPCS or NAICS, to describe the functionality of a Web Service. Although such descriptions are valuable, currently they cannot be directly used in planning. I will conduct an in-depth analysis of the Web Service description characteristics. As a result of this analysis I aim to identify the features that are critical for the automation of the composition task and investigate how these features can be used.

4.2 Planning with Distributed Descriptions

In classical planning, the planner is typically given complete information about the planning domain. The set of all the operators (and methods) that can be used to solve the problem is given to the planner as the input of the planning problem. However, Web Service descriptions will be distributed over the Web, possibly stored in specialized Web Service repositories that use technologies like UDDI. A planner will need to communicate with these remote Web Service registries to find relevant Web Services during the planning process.

When the domain knowledge is distributed over multiple sources, the most important issue to solve is how to integrate Web Services that are supplied by different sources that use possibly distinct vocabularies (ontologies). In HTN planning, when the planner is searching for the possible decompositions of a given task, the methods in the domain knowledge are matched based on the name of a task and its functional signature, i.e. the number of the parameters and their types. This simple matching criteria will obviously fail in a distributed and decentralized environment as separate developers cannot be expected to use the same names for their Web Service descriptions.

When the domain knowledge is distributed over multiple sources, it is required to have more expressive task descriptions in order to match tasks at hand with remote Web Service descriptions. I will investigate how to describe composite services so matching and selection can be done effectively. I will examine two different paradigms for describing composite Web Services:

- Complete Web Service Description: Every step of the composite Web Service is bound to a specific Web Service name. The decomposition of the service is expressed as a collection of existing services combined in a control construct.
- Partial Web Service Description: Some steps of the composite Web Service is not described in terms of concrete actions. Instead these steps have abstract definitions that outline the general features of the service that can be used at this step.

Partial descriptions are very useful when the exact Web Service to accomplish a task is not known at design time. This type of description maximizes the possibility of sharing and reusing Web Service. Therefore, many Web Service description languages allow constructs to model such *partial descriptions*, e.g. abstract processes in BPEL4WS and the SimpleProcess construct in OWL-S. On the other hand, it is easier to generate *complete descriptions* as shown in section 3.1, tools can facilitate this process.

I will investigate how to utilize Web Ontologies to express these two different types of descriptions so that effective task selection and matching can be achieved. My intuition is to exploit the analogy between *partial descriptions* (similarly *complete descriptions*) and classes (instances) in ontologies. Task selection can then be formulated as an ordinary reasoning problem, e.g. the instance retrieval problem. The challenge is to find the right level of expressivity for the Web Service descriptions so that effective matching can be done without the loss of correctness. My aim is to investigate the trade-off between the generality of descriptions and the success of the selection. For example, having a general *BookSellingService* would help us to find a lot of possible matches but most of these matches could be useless because different instances of this category may have very different constraints, e.g. accepting different credit cards, different rules about shipping, etc.

4.3 Planning with Incomplete Information

In the Web context it is not realistic to assume that a planner will have the complete information about the world. The information required to solve a problem needs to be acquired from external sources. Considering that the amount of information available on the Web is huge, the planner should gather the information as needed by the planning process. This means that information-gathering should be interleaved with the planning process.

My preliminary work shows some results on how to interleave the information gathering process with planning. However, there are various restrictions in the preliminary work that need to be addressed. For example, it was assumed that information-providing services cannot have any world-altering effects. Without this assumption, the correctness of the plans generated cannot be guaranteed since the changes done by the information-providing service may invalidate some of the steps the planner has already committed to. For example, paying a fee to acquire some information may invalidate the previous steps that committed the money to other tasks. However, this restriction is not necessary when the effects of the information-providing services do not interact with the plan being sought for. If we consider the previous example, it would be safe to execute the fee-based service and change the state of the world if the original planning problem has nothing to do with money or there is a reasonable budget that is enough for both tasks. I will investigate the ways to relax the restricting assumptions and identify the necessary conditions where world altering information-providing services can safely be executed.

Another important missing piece of the preliminary work is how to find the Web Services which will provide the requested information. For example, in the case of the appointment scheduling example, it is required to find the available time slots for the hospital. In the preliminary work, this knowledge was assumed to somehow exist in the domain description. In the real world, Web Services whose descriptions match the requested query need to be discovered and executed. Actually, it might be necessary to execute a set of Web Services to answer a query. For example, an information providing service may first require you to sign up for the service and supply a username and password to ask a question. Or the query might only be answered by combining information from various different sources. This information-gathering problem itself may be posed as another planning problem where the goal is to generate a plan that will yield the required information upon execution. However, this means that the objective will most typically be a goal formula, which is the case in action-based planning, rather than a task, which is the case in HTN planning. This indicates that combining these two methodologies might be fruitful to solve this problem. I will do further analysis to investigate the applicability of this approach in the Web Services domain.

The information available to the planner is not only the results returned from the Web Services but also the inferences dictated by the Web Ontologies. These two kinds of information should be combined together in order to have a complete understanding of the state. I will investigate how sound and complete reasoning can be done over a set of ontologies and information supplied by Web Services as if there is one single knowledge base.

4.4 Other Issues

In the previous sections, I have outlined the main focus of my research. However, there are various other issues that need to be considered in the context of Web Service composition problem. The following paragraphs briefly discuss these issues.

Interaction with Multiple Agents As stated earlier, most of the time the planner will not have necessary knowledge or enough computational resources to solve the problem at hand. In my future work, I suggest gathering information from remote Web sources to overcome this problem. In this view, remote Web sources are modeled as reactive agents that return answers for given queries. However, in reality, these agents may have more sophisticated capabilities that can be exploited during the composition process. For example, the remote Web Service registry itself may have the ability to create compositions if enough information about the problem is provided. The discovery process can then be done in a more conversational style. Note that interaction with humans can also be modeled this way, e.g. a human user can be represented as another remote agent that the planner can communicate with.

My interest on this subject is on the cooperative aspects of the multi-agent interaction where all the agents are trying to cooperate in some level with each other to accomplish a set of shared or overlapping goals. The level of cooperation between agents may vary depending on the situation. For example, the user who is using the planner to find a composition would be a fully-cooperative agent giving any kind of help to the planning agent. On the other hand, in a B2B application, parties would be less cooperative in the sense that not every participant will be willing to share all the information he/she has. In this scenario, all the parties involved share an overlapping goal, e.g. purchase of a product, but each party has different objective functions, e.g. seller trying to maximize the profit where the buyer is trying to minimize the cost.

Generating Complex Workflows In classical planning, the result of the planning process is typically a totally (or partially) ordered set of operators. In the presence of nondeterminism, the resulting plan may involve conditional branches that contain sensing actions. Generating such conditional plans is crucial for Web Service composition because the information used to generate a composition may very well change at execution time. It is also possible that a Web Service in the plan fails during execution due to an exception. The plan generated needs to be robust enough to handle these cases. For example, undoing the effects of the previous steps of the plan may be required, e.g. a payment order is canceled if the Web Service that arranges shipment has failed.

Composition Analysis and Optimization For any given task, it is probable to find compositions with different components or even compositions with different structure that would achieve the objective. It is not satisfactory to find only the first solution to the composition problem. It is necessary to find all the “promising” compositions and sort these solutions based on an optimality criteria. Of course, this requires the use of some kind of metrics to assign a utility value to a composition. It is not easy to come up with such metrics since there are many different dimensions that need to be considered including the reliability, cost and duration of components in the composition.

References

[1] Semantic Web Services Initiative (SWSI). http://www.swsi.org/.

[2] J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 735–740. Kaufmann, San Mateo, CA, 1990.

[3] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems - Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[4] 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.

[5] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003)*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.

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

[7] C. R. Boualem Benatallah, Mohand-Said Hacid and F. Toumani. Request rewriting-based web service discovery. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, October 2003.

[8] D. Brickley and R. Guha. RDF Vocabulary Description Language: RDF Schema. W3C Recommendation 10 February 2004 http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[9] D. Brickley and R. Guha. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation submitted 22 February 1999 http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[11] G. Collins and L. Pryor. Planning under uncertainty: Some key issues. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1567–1573, 1995.

[12] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0, July 2001. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[13] U. K. D. Roman, H. Lausen. Web service modeling ontology - standard (wsmo - standard). version 0.1 available at http://www.wsmo.org/2004/d2/v0.3/20040329.

[14] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 http://www.w3.org/TR/2002/WD-owl-ref-20021112/.

[15] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Technical Report KSL-03-14, Knowledge Systems Laboratory, Stanford University, CA, 2003.

[16] 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.

[17] M. Fox and D. Long. International planning competition, 2002. http://www.dur.ac.uk/d.p.long/competition.html.

[18] 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.

[19] G. D. Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[20] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. Technical Report TR96-01-09, Department of Computer Science, University of Washington, February 1996.

[21] J. Gonzalez-Castillo, D. Trastour, and C. Bartolini. Description Logics for Matchmaking of Services. In *Workshop on Applications of Description Logics ADL 2001*, Vienna, 2002.

[22] 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.

[23] I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, number 2870 in Lecture Notes in Computer Science, pages 17–29. Springer, 2003.

[24] 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.

[25] I. Horrocks and S. Tessaris. Querying the semantic web: a formal approach. In I. Horrocks and J. Hendler, editors, *Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science, pages 177–191. Springer-Verlag, 2002.

[26] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. In *Proceedings of 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, November 2003.

[27] E. Lambrecht and S. Kambhampati. Planning for information gathering: A survey. Technical Report ASU CSE TR 97-017, Arizona State University, AZ, 1997.

[28] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[29] L. Li and I. Horrocks. A Software Framework For Matchmaking Based on Semantic Web Technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, Budapest, Hungary, May 2003.

[30] 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.

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

[32] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, may 2002.

[33] 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.

[34] 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.

[35] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *The First International Semantic Web Conference*, 2002.

[36] E. P. D. Pednault. ADL: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332. Morgan Kaufmann Publishers Inc., 1989.

[37] J. Peer. Bringing Together Semantic Web and Web Services. In *International Semantic Web Conference 2002 (ISWC'02)*, Italy, 2002.

[38] Pellet. Pellet - OWL DL Reasoner, 2003. http://www.mindswap.org/2003/pellet.

[39] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, pages 103–114. Kaufmann, San Mateo, CA, 1992.

[40] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, pages 103–114. Kaufmann, San Mateo, CA, 1992.

[41] R. Reiter. *Knowledge In Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[42] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. In *Journal of Web Semantics*, To appear.

[43] G. Teege. Making the difference: A subtraction operation for description logics. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the 4th International Conference (KR94)*, San Francisco, CA, 1994. Morgan Kaufmann.

[44] UDDI. The UDDI technical white paper, 2000. http://www.uddi.org/.

[45] W3C. SOAP 1.2 - W3C Recommendation 24 June 2003, 2001. http://www.w3.org/TR/soap12-part0/.

[46] R. Worden. Meaning Definition Language v2.06, July 2002. http://www.charteris.com/XMLToolkit/MDL.asp.

[47] 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.