Web Service Composition Via Generic Procedures and Customizing User Preferences

Shirin Sohrabi, Nataliya Prokoshyna, and Sheila A. McIlraith

Department of Computer Science, University of Toronto, Toronto, Canada
{shirin, nataliya, sheila}@cs.toronto.edu

Abstract. We claim that user preferences are a key component of Web service composition – a component that has largely been ignored. In this paper we propose a means of specifying and intergrating user preferences into Web service composition. To this end, we propose a means of performing automated Web service composition by exploiting generic procedures together with rich qualitative user preferences. We exploit the agent programming language Golog to represent our generic procedures and a first-order preference language to represent rich qualitative temporal user preferences. From these we generate Web service compositions that realize the generic procedure, satisfying the user's hard constraints and optimizing for the user's preferences. We prove our approach sound and optimal. Our system, GologPref, is implemented and interacting with services on the Web. The language and techniques proposed in this paper can be integrated into a variety of approaches to Web or Grid service composition.

1 Introduction

Web services provide a standardized means for diverse, distributed software applications to be published on the Web and to interoperate seamlessly. Simple Web accessible programs are described using machine-processable descriptions and can be loosely composed together to achieve complex behaviour. The weather service at www.weather.com and the flight-booking services at www.aircanada.ca, are examples of Web applications that can be described and composed as Web services. They might be coupled as part of a travel-booking service, for example.

Automated Web service composition is one of many interesting challenges facing the Semantic Web. Given computer-interpretable descriptions of: the task to be performed, the properties and capabilities of available Web services, and possibly some information about the client or user's specific constraints, *automated Web service composition* requires a computer program to automatically select, integrate and invoke multiple Web services in order to achieve the specified task in accordance with any user-specific constraints. Compositions of Web or Grid services are necessary for realizing both routine and complex tasks on the Web (resp. Grid) without the need for time-consuming manual composition and integration of information. Compositions are also a useful way of enforcing business rules and policies in both Web and Grid computing.

I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273, pp. 597–611, 2006.

Fully automated Web service composition has been characterized as akin to both an artificial intelligence (AI) planning task and to a restricted software synthesis task (e.g., [1]). A composition can be achieved using classical AI planning techniques by conceiving services as primitive or complex actions and the task description specified as a (final state) goal (e.g., [2,3]). This approach has its drawbacks when dealing with data. In general, the search space for a composition (aka plan) is huge because of the large number of available services (actions), which grow far larger with grounding for data.

A reasonable middle ground which we originally proposed in [4,1] is to use *generic procedures* to specify the task to be performed and to customize these procedures with *user constraints*. We argued that many of the tasks performed on the Web or on intranets are repeated routinely, and the basic steps to achieving these tasks are well understood, at least at an abstract level – travel planning is one such example. Nevertheless, the realization of such tasks varies as it is tailored to individual users. As such, our proposal was to specify such tasks using a workflow or generic procedure and to customize the procedure with user constraints at run time. Such an approach is generally of the same complexity as planning but the search space is greatly reduced, and as such significantly more efficient than planning without such generic advice.

In [1] we proposed to use an augmented version of the agent programming language Golog [5] to specify our generic procedures or workflows with sufficient nondeterminism to allow for customization. (E.g., *"book inter-city transportation, local transportation and accommodations in any order"*). User constraints (e.g., *"I want to fly with Air Canada."*) were limited to hard constraints (as opposed to "soft"), were specified in first-order logic (FOL), and were applied to the generic procedure at run-time to generate a user-specific composition of services. A similar approach was adopted using hierarchical task networks (HTNs) to represent generic procedures or templates, and realized using SHOP2 (e.g., [6]) without user customization of the procedures.

In this paper, we extend our Golog framework for Web service composition, customizing Golog generic procedures not only with hard constraints but with *soft* user constraints (henceforth referred to as *preferences*). These preferences are defeasible and may not be mutually achievable. We argue that user preferences are a critical and missing component of most existing approaches to Web service composition. User preferences are key for at least two reasons. First, the user's task (specified as a goal and/or generic procedure with user constraints) is often under constrained. As such, it induces a family of solutions. User preferences enable a user to specify properties of solutions that make them more or less desirable. The composition system can use these to generate preferred solutions.

A second reason why user preferences are critical to Web service composition is with respect to *how* the composition is performed. A key component of Web service composition is the selection of specific services used to realize the composition. In AI planning, primitive actions (the analogue of services) are selected for composition based on their preconditions and effects, and there is often only one primitive action that realizes a particular effect. Like actions, services are

selected for composition based on functional properties such as inputs, output, preconditions and effects, but they are also selected based on domain-specific nonfunctional properties such as, in the case of airline ticket booking, whether they book flights with a carrier the user prefers, what credit cards they accept, how trusted they are, etc. By integrating user preferences into Web service composition, preferences over services (the *how*) can be specified and considered along side preferences over the solutions (the *what*).

In this paper we recast the problem of Web service composition as the task of finding a composition of services that achieves the task description (specified as a generic procedure in Golog), that achieves the user's hard constraints, and that is *optimal* with respect to the user's preferences. To specify user preferences, we exploit a rich qualitative preference language, recently proposed by Bienvenu et al. to specify users' preferences in a variant of linear temporal logic (LTL) [7]. We prove the soundness of our approach and the optimality of our compositions with respect to the user's preferences. Our system can be used to select the optimal solution from among families of solutions that achieve the user's stated objective. Our system is implemented in Prolog and integrated with a selection of scraped Web services that are appropriate to our test domain of travel planning.

The work presented here is cast in terms of FOL, *not* in terms of one of the typical Semantic Web languages such as OWL [8] nor more specifically in terms of a semantic Web service ontology such as OWL-S [9], WSMO [10] or SWSO [11]. Nevertheless, it is of direct significance to semantic Web services. As noted in (e.g., [9]) process models, necessary for Web service composition, cannot be expressed in OWL while preserving all and only the intended interpretations of the process model. OWL (and thus OWL-S) is not sufficiently expressive. Further OWL reasoners are not designed for the type of inference necessary for Web service composition. For both these reasons, Web service composition systems generally translate the relevant aspects of service ontologies such as OWL-S into internal representations such as PDDL that are more amenable to AI planning (e.g., [6,12]). Golog served as one of the inspirations for what is now OWL-S [4] and all the OWL-S constructs have translations into Golog [13]. Further, the semantics of the OWL-S process model has been specified in situation calculus [11,14]. Thus, our Golog generic procedures can be expressed in OWL-S and likewise, OWL-S ontologies can be translated into our formalism. We do not have a current implementation of this translation, but it is conceptually straightforward.

2 Situation Calculus and Golog

We use the situation calculus and FOL to describe the functional and nonfunctional properties of our Web services. We use the agent programming language Golog to specify composite Web services and to specify our generic procedures. In this section, we review the essentials of situation calculus and Golog.

The situation calculus is a logical language for specifying and reasoning about dynamical systems [5]. In the situation calculus, the *state* of the world is expressed in terms of functions and relations (fluents) relativized to a particular

situation s, e.g., $F(\boldsymbol{x}, s)$. In this paper, we distinguish between the set of fluent predicates, $\mathcal{F}$, and the set of non-fluent predicates, $\mathcal{R}$, representing properties that do not change over time. A situation s is a *history* of the primitive actions, $a \in \mathcal{A}$, performed from a distinguished initial situation S_0. The function $do(a, s)$ maps a situation and an action into a new situation thus inducing a tree of situations rooted in S_0. Poss(a,s) is true if action a is possible in situation s.

Web services such as the Web exposed application at www.weather.com are viewed as actions in the situation calculus and are described as actions in terms of a situation calculus basic action theory, $\mathcal{D}$. The details of $\mathcal{D}$ are not essential to this paper but the interested reader is directed to [5,14,1] for further details.

Golog [5] is a high-level logic programming language for the specification and execution of complex actions in dynamical domains. It builds on top of the situation calculus by providing Algol-inspired extralogical constructs for assembling primitive situation calculus actions into complex actions (aka *programs*) δ. These complex actions simply serve as constraints upon the situation tree. Complex action constructs include the following:

a — primitive actions	**if** ϕ **then** δ_1 **else** δ_2 – conditionals
$\delta_1; \delta_2$ — sequences	$\delta_1 \vert \delta_2$ — nondeterministic cho ice of actions
$\phi?$ — tests	$\pi(x)\delta$ — nondeterministic choice of arguments
while ϕ **do** δ — while loops	**proc** $P(\boldsymbol{v})$ δ **endProc** — procedure

We also include the construct **anyorder**$[\delta_1, \ldots, \delta_n]$ which allows any permutation of the actions listed. The conditional and while-loop constructs are defined in terms of other constructs. For the purposes of Web service composition we generally treat iteration as finitely bounded by a parameter k. Such finitely bounded programs are called *tree programs*.

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \stackrel{\text{def}}{=} [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$$

$$\textbf{while}_1(\phi)\ \delta \stackrel{\text{def}}{=} \textbf{if } \phi \textbf{ then } \delta \textbf{ endIf}\ ^{1}$$

$$\textbf{while}_k(\phi)\ \delta \stackrel{\text{def}}{=} \textbf{if } \phi \textbf{ then } [\delta;\ \textbf{while}_{k-1}(\phi)\delta] \textbf{ endIf}$$

These constructs can be used to write programs in the language of the domain theory, or more specifically, they can be used to specify both composite Web services and also generic procedures for Web service composition. E.g.[2],

bookAirTicket($\boldsymbol{x}$) ; **if** far **then** bookCar($\boldsymbol{y}$) **else** bookTaxi($\boldsymbol{y}$) **endIf**
bookCar($\boldsymbol{x}$) ; bookHotel($\boldsymbol{y}$).

In order to understand how we modify Golog to incorporate user preferences, the reader must understand the basics of Golog semantics. There are two popular semantics for Golog programs: the original evaluation semantics [5] and a related single-step transition semantics that was proposed for on-line execution of concurrent Golog programs [15]. The transition semantics is axiomatized through

[1] **if-then-endIf** is the obvious variant of **if-then-else-endIf**.

[2] Following convention we will generally refer to fluents in situation-suppressed form, e.g., $at(toronto)$ rather than $at(toronto, s)$. Reintroduction of the situation term is denoted by $[s]$. Variables are universally quantified unless otherwise noted.

two predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$. Given an action theory $\mathcal{D}$, a program δ and a situation s, *Trans* defines the set of possible successor configurations (δ', s') according to the action theory. *Final* defines whether a program successfully terminated, in a given situation. *Trans* and *Final* are defined for every complex action. A few examples follow. (See [15] for details):

$$\begin{aligned}
Trans(nil, s, \delta', s') &\equiv False \\
Trans(a, s, \delta', s') &\equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\
Trans(\phi?, s, \delta', s') &\equiv \phi[s] \wedge \delta' = nil \wedge s' = s \\
Trans([\delta_1; \delta_2], s, \delta', s') &\equiv Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \\
&\quad \vee \exists \delta''. \delta' = (\delta''; \delta_2) \wedge Trans(\delta_1, s, \delta'', s') \\
Trans([\delta_1 \mid \delta_2], s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\pi(x)\delta, s, \delta', s') &\equiv \exists x. Trans(\delta_x^v, s, \delta', s') \\
Final(nil, s) &\equiv \text{TRUE} \qquad Final(a, s) \equiv \text{FALSE} \\
Final([\delta_1; \delta_2], s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s)
\end{aligned}$$

Thus, given the program $bookCar(\boldsymbol{x}); bookHotel(\boldsymbol{y})$, if the action $bookCar(\boldsymbol{x})$ is possible in situation s, then

$$Trans([bookCar(\boldsymbol{x}); bookHotel(\boldsymbol{y})], s, bookHotel(\boldsymbol{y}), do(bookCar(\boldsymbol{x}), s))$$

describes the only possible transition according to the action theory. $do(bookCar(\boldsymbol{x}), s)$ is the transition and $bookHotel(\boldsymbol{y})$ is the remaining program to be executed. Using the transitive closure of *Trans*, denoted $Trans^*$, one can define a *Do* predicate as follows. This *Do* is equivalent to the original evaluation semantics *Do* [15].

$$Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'). \tag{1}$$

Given a domain theory, $\mathcal{D}$ and Golog program δ, program execution must find a sequence of actions $\boldsymbol{a}$ (where $\boldsymbol{a}$ is a vector of actions) such that: $\mathcal{D} \models Do(\delta, S_0, do(\boldsymbol{a}, S_0))$. $Do(\delta, S_0, do(\boldsymbol{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\boldsymbol{a}, S_0)$, where $do(\boldsymbol{a}, S_0)$ abbreviates $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$. Thus, given a generic procedure, described as a Golog program δ, and an initial situation S_0, we would like to infer a terminating situation $do(\boldsymbol{a}, S_0)$ such that the vector $\boldsymbol{a}$ denotes a sequence of Web services that can be performed to realize the generic procedure.

3 Specifying User Preferences

In this section, we describe the syntax of the first-order language we use for specifying user preferences. This description follows the language we proposed in [7] for preference-based planning. The semantics of the language is described in the situation calculus. We provide an informal description here, directing the reader to [7] for further details. Our language is richly expressive, enabling the expression of static as well as temporal preferences. Unlike many preference languages, it provides a total order on preferences. It is qualitative in nature,

facilitating elicitation. Unlike many ordinal preference languages, our language provides a facility to stipulate the relative strength of preferences.

Illustrative example: To help illustrate our preference language, consider the task of travel planning. A generic procedure, easily specified in Golog, might say: *In any order, book inter-city transportation, book local accommodations and book local transportation.* With this generic procedure in hand an individual user can specify their hard constraints (e.g., *Lara needs to be in Chicago July 29-Aug 5, 2006.*) together with a list of preferences described in the language to follow.

To understand the preference language, consider the composition we are trying to generate to be a situation – a sequence of actions or Web services executed from the initial situation. A user specifies his or her preferences in terms of a single, so-called *General Preference Formula*. This formula is an aggregation of preferences over constituent properties of situations (i.e., compositions). The basic building block of our preference formula is a *Basic Desire Formula* which describes properties of (partial) situations (i.e., compositions).

Definition 1 (Basic Desire Formula (BDF)). *A basic desire formula is a sentence drawn from the smallest set $\mathcal{B}$ where:*

1. $\mathcal{F} \subset \mathcal{B}$
2. $\mathcal{R} \subset \mathcal{B}$
3. *If* $f \in \mathcal{F}$, *then* $\boldsymbol{final}(f) \in \mathcal{B}$
4. *If* $a \in \mathcal{A}$, *then* $\boldsymbol{occ}(a) \in \mathcal{B}$
5. *If* φ_1 *and* φ_2 *are in* $\mathcal{B}$, *then so are* $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $(\exists x)\varphi_1$, $(\forall x)\varphi_1$, $\boldsymbol{next}(\varphi_1)$, $\boldsymbol{always}(\varphi_1)$, $\boldsymbol{eventually}(\varphi_1)$, *and* $\boldsymbol{until}(\varphi_1, \varphi_2)$.

$\mathbf{final}(f)$ states that fluent f holds in the final situation, $\mathbf{occ}(a)$ states that action a occurs in the present situation, and $\mathbf{next}(\varphi_1)$, $\mathbf{always}(\varphi_1)$, $\mathbf{eventually}(\varphi_1)$, and $\mathbf{until}(\varphi_1, \varphi_2)$ are basic LTL constructs.

BDFs establish properties of preferred situations (i.e., compositions of services). By combining BDFs using boolean connectives we are able to express a wide variety of properties of situations. E.g.[3]

$$\mathbf{final}(at(home)) \tag{P1}$$

$$(\exists\, \boldsymbol{c}).\mathbf{occ}'(bookAir(\boldsymbol{c}, economy, direct)) \wedge member(\boldsymbol{c}, starAlliance) \tag{P2}$$

$$\mathbf{always}(\neg((\exists\, \boldsymbol{h}).hotelBooked(\boldsymbol{h}) \wedge hilton(\boldsymbol{h}))) \tag{P3}$$

$$(\exists\, \boldsymbol{h}, \boldsymbol{r}).(\mathbf{occ}'(bookHotel(\boldsymbol{h}, \boldsymbol{r})) \wedge paymentOption(\boldsymbol{h}, visa) \wedge starsGE(\boldsymbol{r}, 3) \tag{P4}$$

P1 says that in the final situation Lara prefers to be at home. P2 says that Lara prefers to eventually book direct economy air travel with a Star Alliance carrier. Recall there was no stipulation in the generic procedure regarding the mode of transportation between cities or locally. P3 expresses the preference

[3] To simplify the examples many parameters have been suppressed. For legibility, variables are bold faced, we abbreviate $\mathbf{eventually}(\mathbf{occ}(\varphi))$ by $\mathbf{occ}'(\varphi)$, and we refer to the preference formulae by their labels.

that a Hilton hotel never be booked while P4 expresses a preference for hotels that accept visa credit cards and have a rating of 3 stars or more.

To define a preference ordering over alternative properties of situations, we define *Atomic Preference Formulae* (APFs). Each alternative being ordered comprises 2 components: the property of the situation, specified by a BDF, and a *value* term which stipulates the relative strength of the preference.

Definition 2 (Atomic Preference Formula (APF)). *Let $\mathcal{V}$ be a totally ordered set with minimal element v_{min} and maximal element v_{max}. An atomic preference formula is a formula $\varphi_0[v_0] \gg \varphi_1[v_1] \gg ... \gg \varphi_n[v_n]$, where each φ_i is a BDF, each $v_i \in \mathcal{V}$, $v_i < v_j$ for $i < j$, and $v_0 = v_{min}$. When $n = 0$, atomic preference formulae correspond to BDFs.*

An APF expresses a preference over alternatives. In what follows, we let $\mathcal{V} = [0, 1]$, but we could instead choose a strictly qualitative set like $\{best < good < indifferent < bad < worst\}$ since the operations on these values are limited to max and min. The following APFs express an ordering over Lara's preferences.

$$\begin{aligned}
&\text{P2}[0] \\
&\quad \gg (\exists\, \boldsymbol{c}, \boldsymbol{w}).\mathbf{occ}'(bookAir(\boldsymbol{c}, economy, \boldsymbol{w}) \wedge member(\boldsymbol{c}, starAlliance))[0.2] \\
&\qquad \gg \mathbf{occ}'(bookAir(delta, economy, direct))[0.5] && \text{(P5)} \\
&(\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(national, \boldsymbol{t}))[0] \gg (\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(alamo, \boldsymbol{t}))[0.2] \\
&\quad \gg (\exists\, \boldsymbol{t}).\mathbf{occ}'(bookCar(avis, \boldsymbol{t}))[0.8] && \text{(P6)} \\
&(\exists\, \boldsymbol{c}).\mathbf{occ}'(bookCar(\boldsymbol{c}, suv))[0] \gg (\exists\, \boldsymbol{c}).\mathbf{occ}'(bookCar(\boldsymbol{c}, compact))[0.2] && \text{(P7)}
\end{aligned}$$

P5 states that Lara prefers direct economy flights with a Star Alliance carrier, followed by economy flights with a Star Alliance carrier, followed by direct economy flights with Delta airlines. P6 and P7 are preference over cars. Lara strongly prefers National and then Alamo over Avis, followed by any other car-rental companies. Finally she slightly prefers an SUV over a compact with any other type of car a distant third.

To allow the user to specify more complex preferences and to aggregate preferences, General Preference Formulae (GFPs) extend our language to conditional, conjunctive, and disjunctive preferences.

Definition 3 (General Preference Formula (GPF)). *A formula Φ is a general preference formula if one of the following holds:*

- *Φ is an APF*
- *Φ is $\gamma : \Psi$, where γ is a BDF and Ψ is a GPF [Conditional]*
- *Φ is one of*
  - *$\Psi_0 \,\&\, \Psi_1 \,\&\, ... \,\&\, \Psi_n$ [General Conjunction]*
  - *$\Psi_0 \mid \Psi_1 \mid ... \mid \Psi_n$ [General Disjunction]*

where $n \geq 1$ and each Ψ_i is a GPF.

Continuing our example:

$$(\forall \boldsymbol{h}, \boldsymbol{c}, \boldsymbol{e}, \boldsymbol{w}).\mathbf{always}(\neg hotelBooked(\boldsymbol{h}) : \neg \mathbf{occ}'(bookAir(\boldsymbol{c}, \boldsymbol{e}, \boldsymbol{w}))) \quad \text{(P8)}$$

$$far \ : \ \text{P5} \quad \text{(P9)}$$

$$\text{P3} \,\&\, \text{P4} \,\&\, \text{P6} \,\&\, \text{P7} \,\&\, \text{P8} \,\&\, \text{P9} \quad \text{(P10)}$$

P8 states that Lara prefers not to book her air ticket until she has a hotel booked. P9 conditions Lara's airline preferences on her destination being far away. (If it is not far, she will not fly and the preferences are irrelevant.) Finally, P10 aggregates previous preferences into one formula.

Semantics: Informally, the semantics of our preference language is achieved through assigning a weight to a situation s with respect to a GPF, Φ, written $w_s(\Phi)$. This weight is a composition of its constituents. For BDFs, a situation s is assigned the value v_{min} if the BDF is satisfied in s, v_{max} otherwise. Recall that in our example above $v_{min} = 0$ and $v_{max} = 1$, though they could equally well have been a qualitative e.g., [excellent, abysmal]. Similarly, given an APF, and a situation s, s is assigned the weight of the best BDF that it satisfies within the defined APF. Returning to our example above, for P6 if a situation (composition) booked a car from Alamo rental car, it would get a weight of 0.2. Finally GPF semantics follow the natural semantics of boolean connectives. As such General Conjunction yields the maximum of its constituent GPF weights and General Disjunction yields the minimum of its constituent GPF weights. For a full explanation of the situation calculus semantics, please see [7]. Here we also define further aggregations that can be performed. These are mostly syntactic sugar that are compelling to the user and we omit them for space.

We conclude this section with the following definition which shows us how to compare two situations (and thus two compositions) with respect to a GPF:

Definition 4 (Preferred Situations). *A situation s_1 is at least as preferred as a situation s_2 with respect to a GPF Φ, written $pref(s_1, s_2, \Phi)$ if $w_{s_1}(\Phi) \leq w_{s_2}(\Phi)$.*

4 Web Service Composition

In this section, we define the notion of web service composition with generic procedures and customizing user preferences, present an algorithm for computing these compositions and prove properties of our algorithm. Our definition relies on the definition of *Do* from (1) in Section 2.

Definition 5 (Web Service Composition w/User Preferences (WSCP)). *A Web service composition problem with user preferences is described as a 5-tuple $(\mathcal{D}, O, \delta, C, \Phi)$ where:*

- *$\mathcal{D}$ is a situation calculus basic action theory describing functional properties of the Web services,*
- *O is a FOL theory describing the non-functional properties of the Web services*[4]*,*

[4] The content of $\mathcal{D}$ and O would typically come from an OWL-S, SWSO, or other semantic Web service ontology.

- δ *is a generic procedure described in Golog,*
- C *is a formula expressing hard user constraints, and*
- Φ *is a GPF describing user preferences.*

A Web Service Composition (WSC) is a sequence of Web services $\boldsymbol{a}$ *such that*

$$\mathcal{D} \wedge O \models \exists s.Do(\delta, S_0, s) \wedge s = do(\boldsymbol{a}, S_0) \wedge C(s)$$

A preferred WSC (WSCP) is a sequence of Web services $\boldsymbol{a}$ *such that*

$$\begin{aligned} \mathcal{D} \wedge O \models \exists s.Do(\delta, S_0, s, \Phi) \wedge s = do(\boldsymbol{a}, S_0) \wedge C(s) \\ \wedge \quad \nexists s'.[Do(\delta, S_0, s', \Phi) \wedge C(s') \wedge pref(s', s, \Phi)] \end{aligned}$$

I.e., a WSC is a sequence of Web services, $\boldsymbol{a}$, whose execution starting in the initial situation enforces the generic procedure and hard constraints terminating successfully in $do(\boldsymbol{a}, s)$. A WSCP yields a most preferred terminating situation.

4.1 Computing Preferred Compositions

A Golog program places constraints on the situation tree that evolves from S_0. As such, any implementation of Golog is effectively doing planning in a constrained search space, searching for a legal termination of the Golog program. The actions that define this terminating situation are the plan. In the case of composing web services, this plan is a web service composition.

To compute a preferred composition, WSCP, we search through this same constrained search space to find the *most preferred* terminating situation. Our approach, embodied in a system called GologPref, searches for this optimal terminating situation by modifying the PPLAN approach to planning with preferences proposed in [7]. In particular, GologPref performs best-first search through the constrained search space resulting from the Golog program, $\delta; C$. The search is guided by an admissible evaluation function that evaluates partial plans with respect to whether they satisfy the preference formula, Φ. The admissible evaluation function is the optimistic evaluation of the preference formula, with the pessimistic evaluation and the plan length used as tie breakers where necessary, in that order.

The preference formula is evaluated over intermediate situations (partial compositions) by exploiting *progression* as described in [7]. Informally, progression takes a situation and a temporal logic formula (TLF), evaluates the TLF with respect to the state of the situation, and generates a new formula representing those aspects of the TLF that remain to be satisfied in subsequent situations.

Fig 1 provides a sketch of the basic GologPref algorithm following from PPLAN . The full GologPref algorithm takes as input a 5-tuple $(\mathcal{D}, O, \delta, C, \Phi)$. For ease of explication, our algorithm sketch in Fig 1 explictly identifies the initial situation of $\mathcal{D}$, *init*, the Golog program, $\delta; C$ which we refer to as *pgm* and Φ, which we refer to as *pref*. GologPref returns a sequence of Web services, i.e. a plan, and the weight of that plan. The *frontier* is a list of nodes of the form [*optW*, *pessW*, *pgm*, *partialPlan*, *state*, *pref*], sorted by optimistic weight, pessimistic weight, and then by length. The frontier is initialized to the input program and the empty partial

GologPref(*init*, *pgm*, *pref*)
frontier ← **initFrontier**(*init*, *pgm*, *pref*)
while *frontier* ≠ ∅
  current ← **removeFirst**(*frontier*)
  % establishes current values for *progPgm*, *partialPlan*, *state*, *progPref*
  if *progPgm*=*nil* and *optW*=*pessW*
    return *partialPlan*, *optW*
  end if
  neighbours ← **expand**(*progPgm*, *partialPlan*, *state*, *progPref*)
  frontier ← **sortNmergeByVal**(*neighbours*, *frontier*)
end while
return [], ∞

expand(*progPgm*, *partialPlan*, *state*, *progPref*) returns a list of new nodes to add to the frontier. If *partialPlan*=*nil* then **expand** returns [ ]. Otherwise, **expand** uses Golog's *Trans* to determine all the executable actions that are legal transitions of *progPgm* in *state* and to compute the remaining program for each.
It returns a list which contains, for each of these executable actions *a* a node
  (*optW*, *pessW*,*newProgPgm*, *newPartialPlan*, *newState*, *newProgPref*)
and for each *a* leading to a terminating state, a second node
  (*realW*, *realW*, *nil*, *newPartialPlan*, *newState*, *newProgPref*).

Fig. 1. A sketch of the GologPref algorithm

plan, its *optW*, *pessW*, and *pref* corresponding to the progression and evaluation of the input preference formula in the initial state.

On each iteration of the **while** loop, GologPref removes the first node from the frontier and places it in *current*. If the Golog program of *current* is *nil* then the situation associated with this node is a terminating situation. If it is also the case that *optW*=*pessW*, then GologPref returns *current*'s partial plan and weight. Otherwise, it calls the function **expand** with *current's* node as input.

expand returns a new list of nodes to add to the frontier. If *progPgm* is *nil* then no new nodes are added to the frontier. Otherwise, **expand** generates a new set of nodes of the form [*optW*, *pessW*, *prog*, *partialPlan*, *state*, *pref*], one for each action that is a legal Golog transition of *pgm* in *state*. For actions leading to terminating states, **expand** also generates a second node of the same form but with *optW* and *pessW* replaced by the actual weight achieved by the plan. The new nodes generated by **expand** are then sorted by *optW*, *pessW*, then length and merged with the remainder of the frontier. If we reach the empty frontier, we exit the **while** loop and return the empty plan.

We now prove the correctness of our algorithm.

Theorem 1 (Soundness and Optimality). *Let* $\mathcal{P}=(\mathcal{D}, O, \delta, C, \Phi)$ *be a Web service composition problem, where* δ *is a tree program. Let* $\boldsymbol{a}$ *be the plan returned by GologPref from input* $\mathcal{P}$*. Then* $\boldsymbol{a}$ *is a WSCP of* $(\mathcal{D}, O, \delta, C, \Phi)$.

Proof sketch: We prove that the algorithm terminates appealing to the fact that δ is a tree program. Then we prove that $\boldsymbol{a}$ is a WSC by cases over *Trans* and *Final*. Finally we prove that $\boldsymbol{a}$ is also optimal, by exploiting the correctness of progression of preference formuale proven in [7], the admissibility of our evaluation function, and the bounded size of the search space generated by the Golog program $\delta; C$.

4.2 Integrated Optimal Web Service Selection

Most Web service composition systems use AI planning techniques and as such generally ignore the important problem of Web service selection or discovery, assuming it will be done by a separate matchmaker. The work presented here is significant because it enables the selection of services for composition based, not only on their inputs, outputs, preconditions and effects but also based on other nonfunctional properties. As such, users are able to specify properties of services that they desire along side other properties of their preferred solution, and services are selected that optimize for the users preferences in the context of the overall composition.

To see how selection of services can be encoded in our system, we reintroduce the service parameter $\boldsymbol{u}$ which was suppressed from the example preferences in Section 3. Revisiting P2, we see how the selection of a service $\boldsymbol{u}$ is easily realized within our preference framework with preference P2'.

$$(\exists\ \boldsymbol{c}, \boldsymbol{u}).\mathbf{occ}'(bookAir(\boldsymbol{c}, economy, direct, \boldsymbol{u})) \wedge member(\boldsymbol{c}, starAlliance) \\ \wedge\ serviceType(\boldsymbol{u}, airTicketVendor) \wedge sellsTickets(\boldsymbol{u}, \boldsymbol{c}) \qquad \text{(P2')}$$

5 Implementation and Application

We have implemented the generation of Web Service compositions using generic procedures and customizing user preferences as described in previous sections. Our implementation, GologPref, builds on an implementation of PPLAN [7] and an implementation of IndiGolog [5] both in SWI Prolog[5].

GologPref interfaces with Web services on the Web through the implementation of domain-specific scrapers developed using AgentBuilder 3.2, and AgentRunner 3.2, Web agent design applications developed by Fetch Technologies ©. Among the sites we have scraped are Mapquest, and several air, car and hotel services. The information gathered is collected in XML and then processed by GologPref.

We tested GologPref in the domain of travel planning. Our tests serve predominantly as a proof of the concept and to illustrate the utility of GologPref.

Our generic procedure which is represented in Golog was very simple, allowing flexibility in how it could be instantiated. What follows is an example of the Prolog encoding of a GologPref generic procedure.

[5] See [5] for a description of the translation of $\mathcal{D}$ to Prolog.

```
anyorder [bookAcc, bookCityToCityTranspo, bookLocalTranspo]

proc(bookAcc(Location, Day, Num),
[ stayWithFriends(Location) | bookHotel(Location, Day, Num) ]).

proc(bookLocalTranspo(Location, StartDay, ReturnDay),
[       getRide(Location, StartDay, ReturnDay)  |
        walk(Location)   |   bookCar(Location, StartDay, ReturnDay)  ]).

proc(bookCityToCityTranspo(Location, Des, StartDay, ReturnDay),
[       getRide(Location, Des, StartDay, ReturnDay) |
        bookAir(Location, Des, StartDay, ReturnDay) |
        bookCar(Location, Des, StartDay, ReturnDay) ]).
```

We tested our GologPref generic procedure with 3 different user profiles: Jack the impoverished university student, Lara the picky frequent flyer, and Conrad the corporate executive who likes timely luxury travel. Each user lived in Toronto and wanted to be in Chicago for specific days. A set of rich user preferences were defined for each user along the lines of those illustrated in Section 3. These preferences often required access to different Web information, such as driving distances. Space precludes listing of the preferences, code and full test results, but these are available at `http://www.cs.toronto.edu/~sheila/gologpref/`.

Not surprisingly, in all cases, GologPref found the optimal WSC for the user. Compositions varied greatly ranging from Jack who arranged accommodations with friends; checked out the distance to his local destinations and then arranged his local transportation (walking since his local destination was close to where he was staying); then once his accommodations were confirmed, booking an economy air ticket Toronto-Chicago with one stop on US Airways with Expedia. Lara on the other hand, booked a hotel (not Hilton), booked an intermediate-sized car with National, and a direct economy air ticket with Star Alliance partner Air Canada via the Air Canada Web site. The optimality and the diversity of the compositions, all from the same generic procedure, illustrate the flexibility afforded by the WSCP approach.

Figure 2 shows the number of nodes expanded relative to the search space size for 6 test scenarios. The full search space represents all possible combinations of city-to-city transportation, accommodations and local transportation available to the users which could have been considered. These results illustrate the effectiveness of the heuristic used to find optimal compositions.

6 Summary and Related Work

In this paper we argued that the integration of user preferences into Web service composition was a key missing component of Web service composition. Building on our previous framework for Web service composition via generic procedures [1] and our more recent work on preference-based planning [7], we proposed a system for Web service composition with user preferences. Key contributions of this paper include: characterization of the task of Web service composition with

Case Number	Nodes Expanded	Nodes Considered	Time (sec)	Nodes in Full Search Space
1	104	1700	20.97	28,512
2	102	1647	19.93	28,512
3	27	371	2.88	28,512
4	27	368	2.92	28,512
5	99	1692	21.48	28,512
6	108	1761	21.29	28,512

Fig. 2. Test results for 6 scenarios run under Windows XP with a 593MHz processor and 512 MB of RAM. The times shown are five run averages.

generic procedures and user preferences, provision of a previously developed language for specifying user preferences, provision of the GologPref algorithm that integrates preference-based reasoning into Golog, a proof of the soundness and optimality of GologPref with respect to the user's preferences, and a working implementation of our GologPref algorithm. A notable side effect of our framework is the seamless integration of Web service selection with the composition process.

We tested GologPref on 6 diverse scenarios applied to the same generic procedure. Results illustrated the diversity of compositions that could be generated from the same generic procedure. The number of nodes expanded by the heuristic search was several orders of magnitude smaller than the grounded search space, illustrating the effectiveness of the heuristic and the Golog program in guiding search.

A number of researchers have advocated using AI planning techniques to address the task of Web service composition including using regression-based planners [2], planners based on model checking (e.g., [3]), highly optimized hierarchical task network (HTN) planners such as SHOP2 (e.g., [16]), and most recently a combination of classical and HTN planning called XPLAN [12]. Like Golog, HTNs afford the user the ability to define a generic procedure or *template* of how to perform a task.

Recently Sirin et al. incorporated simple service preferences into the SHOP2 HTN planner to achieve dynamic service binding [6]. Their preference language is significantly less expressive than the one presented here and is restricted to the task of service selection rather than solution optimization. Nevertheless, it is a promising start. The most related previous work was performed by Fritz and the third author in which they *precompiled* a subset of the preference language presented here into Golog programs that were then integrated with a decision-theoretic Golog (DTGolog) program [17]. The main objective of this work was to provide a means of integrating qualitative and quantitative preferences for agent programming. While both used a form of Golog, the form and processing of preferences was quite different. We know of no other work integrating preferences into Web service composition. Nevertheless, there is a recent focus on preference-based planning. Early preference-based planners include PPLAN [7]

and an approach to preference-based planning using answer set programming [18]. A number of preference-based planners were developed for the 2006 International Planning Competition (IPC-5) and are yet to be published. Preliminary descriptions of these planners can be found at `http://zeus.ing.unibs.it/ipc-5/`.

Acknowledgements

Thanks to Meghyn Bienvenu for her work on PPLAN which was fundamental to the realization of this work. Thanks to Christian Fritz for useful discussions and to Fetch Technologies for allowing us to use their AgentBuilder software. We also gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) and the CRA's Canadian Distributed Mentorship Project (CDMP) for partially funding this research.

References

1. McIlraith, S., Son, T.C.: Adapting Golog for composition of semantic web services. In: Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR02), Toulouse, France (2002) 482–493
2. McDermott, D.V.: Estimated-regression planning for interactions with web services. In: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02). (2002) 204–211
3. Traverso, P., Pistore, M.: Automatic composition of semantic web services into executable processes. In: Proceedings of the Third International Semantic Web Conference (ISWC2004). (2004)
4. McIlraith, S., Son, T., Zeng, H.: Semantic Web services. In: IEEE Intelligent Systems (Special Issue on the Semantic Web). Volume 16. (2001)
5. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge, MA (2001)
6. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP2. Journal of Web Semantics **1**(4) (2005) 377–396
7. Bienvenu, M., Fritz, C., McIlraith, S.: Planning with qualitative temporal preferences. In: Proceedings of the Tenth International Conference on Knowledge Representation and Reasoning (KR06). (2006) 134–144
8. Horrocks, I., Patel-Schneider, P., van Harmelen, F.: From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. Journal of Web Semantics **1**(1) (2003) 7–26
9. Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., McGuinness, D., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. World Wide Web Journal (2006) To appear.
10. Bruijn, J.D., Lausen, H., Polleres, A., Fensel, D.: The web service modeling language WSML: An overview. Technical report, DERI (2006)
11. Battle, S., Bernstein, A., Boley, H., Grosof, B., and R. Hull, M.G., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., Tabet, S.: Semantic web service ontology (SWSO) first-order logic ontology for web services (FLOWS) (2005) http://www.daml.org/services/swsl/report/.

12. Klusch, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with OWLS-Xplan. In: Working notes of the AAAI-05 Fall Symposium on Agents and the Semantic Web, Arlington VA, USA (2005)
13. McIlraith, S.A., Fadel, R.: Planning with complex actions. In: 9th International Workshop on Non-Monotonic Reasoning (NMR), Toulouse, France (2002) 356–364
14. Narayanan, S., McIlraith, S.: Simulation, verification and automated composition of web services. In: Proceedings of the Eleventh International World Wide Web Conference (WWW-2002). (2002)
15. de Giacomo, G., Lespérance, Y., Levesque, H.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121**(1–2) (2000) 109–169
16. Sirin, E., Parsia, B., Hendler, J.: Template-based composition of semantic web services. In: Working notes of the AAAI-05 Fall Symposium on Agents and the Semantic Web. (2005)
17. Fritz, C., McIlraith, S.: Decision-theoretic GOLOG with qualitative preferences. In: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR06). (2006)
18. Son, T., Pontelli, E.: Planning with preferences using logic programming (2007) Theory and Practice of Logic Programming. To appear.