Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning

Erick Martínez and **Yves Lespérance**
Department of Computer Science
York University
Toronto, Ontario M3J 1P3
Canada
{erickm, lesperan}@cs.yorku.ca

Abstract

Motivated by the problem of automated Web service composition (WSC), in this paper, we present some empirical evidence to validate the effectiveness of using knowledge-based planning techniques for solving WSC problems. In our experiments we utilize the PKS (Planning with Knowledge and Sensing) planning system which is derived from a generalization of STRIPS. In PKS, the agent's (incomplete) knowledge is represented by a set of databases and actions are modelled as revisions to the agent's knowledge state rather than the state of the world. We argue that, despite the intrinsic limited expressiveness of this approach, typical WSC problems can be specified and solved at the knowledge level. We show that this approach scales relatively well under changing conditions (e.g. user constraints). Finally, we discuss implementation issues and propose some architectural guidelines within the context of an agent-oriented framework for inter-operable, intelligent multi-agent systems for WSC and provisioning.

Introduction

Web services are self-contained, self-described, active, modular software applications that can be advertised, discovered, and invoked over the Web (Gottschalk & IBM Team 2000), e.g., an airline travel service or a book-buying service. In this paper, we are mainly motivated by the problem of automated Web service composition (henceforth WSC), which can be stated as follows: given a set of Web services and some user-defined task or goal to be achieved, a computer agent should be able to automatically find a composition of the available services to accomplish the task (McIlraith & Son 2002). WSC can either be seen as a software/plan synthesis problem or as a plan execution problem. Planning is computationally demanding, particularly under conditions of incomplete knowledge and sensing. Many actions performed by existing Web services are precisely sensing actions. To tackle WSC as a plan synthesis problem, planning can be performed using predefined available services as the building blocks of a plan. In (McIlraith & Son 2002) an extension of the logic programming language Golog is proposed to address the WSC problem through the provision of high-level generic procedures and user defined customization constraints. Also, a middle-ground approach to execution is defined that combines on-line execution of necessary information-gathering Web services with offline simulation of world-altering services. This approach operates under the assumption of reasonable persistence of sensed information and complete independence between sensing and world-altering actions. But there are scenarios where these assumptions do not hold, e.g, a Web service that requires users to register/login prior to browsing resources.

Some work has addressed the problem of planning under conditions of incomplete knowledge and sensing, for instance, the PKS planner (Petrick & Bacchus 2002) that uses a generalization of the STRIPS (Fikes & Nilsson 1971) approach, and the planner discussed in (Bertoli *et al.* 2001), that uses a model checking approach. One of the features that makes PKS interesting for WSC, is its ability to generate parameterized conditional plans under such conditions. In addition, PKS (as opposed to other approaches based on possible worlds reasoning) generates plans at the knowledge level without considering all the different ways the physical world can be configured. Previous experimental results are also encouraging. In this paper, we explore the applicability of PKS to the automated WSC problem.

The rest of the paper is organized as follows. First, we review the PKS planning system. Next, we discuss a general approach to specifying/solving WSC problems at the knowledge-level using PKS. After that, we focus on representational issues. In particular, we discuss how to keep our specification as generic, customizable, and concise as possible. We also examine the details of some travel domain examples. Then, we present some preliminary experimental results to evaluate the performance and correctness of PKS on these examples. In the following section, we sketch an infrastructure and toolkit for an agent-oriented framework for inter-operable, intelligent multi-agent systems for WSC and provisioning. We also try to situate the experiments within the context of such a framework. Finally, we review what has been achieved and discuss future work.

PKS

PKS (Bacchus & Petrick 1998; Petrick & Bacchus 2002; 2003) is a knowledge-based planning system derived from a

generalization of STRIPS. In STRIPS, the state of the world is represented by a database and actions are represented as updates to that database. The PKS system uses, instead of a single database, a set of databases that represent the agent's knowledge rather than the state of the world. Actions are modelled as knowledge-level modifications to the agent's knowledge and specified as updates to these databases.

Databases: There are four available databases, each one storing a different type of knowledge. The contents of these databases have a fixed formal interpretation in a first-order modal logic of knowledge that characterizes the agent's knowledge state.

K_f : This database can contain any ground literal. In particular, it can store positive and negative facts known to the agent. K_f can also contain formulas specifying knowledge of the value of a function on fixed arguments. The closed world assumption does not apply.

K_w : This database stores formulas whose truth value the agent knows. In particular, K_w can contain any conjunction of ground atomic formulas. Intuitively, $K_w(\alpha)$ means that at planning time we have that either the agent knows α or it knows $\neg\alpha$. The agent will only resolve this disjunction at execution time. This database is used for plan time modelling of the effects of sensing actions.

K_v : This database contains information about function values. In particular, K_v can store unnested function terms whose values are known to the agent at execution time. K_v is used for plan time modelling of the effect of sensing actions that return numeric values.

K_x : This database contains information about disjunctive (exclusive *or*) knowledge of ground literals of the form $(l_1|l_2|...|l_n)$. Intuitively, this formula represents the fact that the agent knows exactly one of the l_i is true.

Note that the only forms of incomplete knowledge that can be expressed are complete lack of knowledge about an atom, by leaving it out of K_f, and knowledge that only one of a finite set of literals are true using K_x. There is no reasoning by cases other than by going through a set of cases that have been explicitly enumerated.

Goals: Simple goals can be represented as primitive queries. A primitive query can take one of the following forms: (i) $K(\alpha)$, is α known to be true? (ii) $K(\neg\alpha)$, is α known to be false? (iii) $K_w(\alpha)$, does the agent know whether α? (iv) $K_v(t)$, does the agent know the value of t? (v) the negation of any of the previous queries. In the above, α represents any ground atomic formula, and t represents any variable free term. Complex goals can be expressed as queries which include primitive queries, conjunctions of queries, disjunctions of queries, and quantified queries where the quantification ranges over the set of known objects.

Actions: Actions are specified in terms of three components: parameters, preconditions, and effects. For example in Table 1, the specifications for actions $open(r)$ and $search(r)$ are given. The former is a physical action to open (the door to) a room. The later is a knowledge-producing action that senses for the presence of a person in a room. Action $open(r)$ requires the agent to know that room r is not already open as a precondition. The effect of $open(r)$ is modelled by the addition of the new fact $opened(r)$ to the K_f database. Action $search(r)$ requires, as a precondition, the agent to know that room r is already open and not knowing about whether someone has been found yet in room r. The effect of $search(r)$ is that the agent comes to know whether someone was found in room r. This is modelled by adding a new literal $found(r)$ to the K_w database. Note that the preconditions in this case are a conjunction of primitives queries. Actions' effects are specified as a set of database updates, some of which can be conditional.

Action	Precondition	Effects
$open(r)$	$K(room(r))$ $K(\neg opened(r))$	$add(K_f, opened(r))$
$search(r)$	$K(room(r))$ $\neg K_w(found(r))$ $K(opened(r))$	$add(K_w, found(r))$

Table 1: $open$ and $search$ actions.

Domain Specific Update Rules
$K(room(r)) \wedge (K(found(r)) \Rightarrow add(K_f, done)$

Table 2: DSUR example.

Domain specific update rules (DSUR): These rules are used to specify additional action effects and correspond to state invariants at the knowledge level. In any knowledge state, DSURs may be triggered provided their conditions are satisfied. A DSUR example is given in Table 2. This rule captures the additional action effect of marking the search task as finished when someone is found in any room. Note that the antecedent of a DSUR can be any goal formula, and the consequent must be a set of database updates.

Planning problems: A planning problem in PKS is defined as a tuple $\langle I, A, U, G\rangle$, where I is the initial state, A is a nonempty set of action specifications, U is a set of DSURs, and G is a goal condition.

The PKS system relies on an efficient, but incomplete, inference algorithm that uses a forward chaining approach to find plans (Bacchus & Petrick 1998). PKS's efficiency comes as a result of its limited expressiveness with respect to the kinds of incomplete knowledge that can be represented. The current implementation supports both undirected depth-first search and breadth-first search (used to find shortest plans).

Composing Web Services in PKS

One of the features that makes the PKS planning system attractive, for the automated WSC task, is its ability to generate parameterized conditional plans (containing run-time variables) in the presence of incomplete knowledge and sensing. Also, plans are generated at the knowledge level without considering all the different ways the physical world can be configured and changed by actions. In (Petrick & Bacchus 2002) some experiments are described which show "impressive performance" on generating plans in a number of classic planning domains and problems (e.g., bomb in the

toilet, medicate, opening a safe and Unix domains). The authors also claim that, for many common problems, this knowledge-based representation "scales better and supports features that makes it applicable to much richer domains".

The main motivation for our experiments is to test both the applicability and scalability of this approach to the WSC problem. In this paper, we primarily address WSC as a plan synthesis problem. Therefore, we do not discuss other issues like plan execution and contingency recovery. Given a particular domain specification and a description of some user defined goals and explicit preferences, we use the PKS planning system to generate conditional plans that solve the task. In our approach each PKS primitive action corresponds to an available service. In particular, knowledge-producing actions correspond to information-gathering services and physical actions to world-altering services. We think that, in principle, this approach is modular and flexible. If a new Web service becomes available we can add it as a new primitive action to the domain specification. Moreover, we can handle cases that previous approaches (McIlraith & Son 2002) cannot, e.g., physical actions having a direct effect on sensing actions. Also, we do not need to have pre-specified generic plans. However, as can be expected with offline simulation, our search space is likely to be large as our conditional plans should cover all the possible alternatives.

Representing Web Services in PKS

One motivation in these experiments is to keep our specifications as general, customizable and intuitive as possible, so that they can be reused by different users under changing conditions. Given the current representational restrictions of PKS, we often cannot represent the user's objective as a goal formula alone, e.g. a general non-exclusive disjunctive goal of the form $K(P(\vec{x}) \vee Q(\vec{x}))$. Nevertheless, one can often get around this limitation by introducing a new fluent for the goal and adding DSURs that make it true under appropriate conditions, e.g. when one of the disjuncts holds. We will show some of this in the BBF and BPBF problems.

We address the generic representation issue by keeping the action specifications as decoupled as possible from the specifics of the goal of a planning problem. We also want to be able to plug in new available services by simply adding the corresponding primitive actions to the domain specification. By separating all goal specific action effects, we get a more modular and generic representation to meet the requirements of different users.

To make our specification accommodate user constraints/preferences, we adopt some of the ideas of (McIlraith & Son 2002). For each action a_i we introduce a fluent $desA_i$ that encapsulates the *necessary conditions* that make a_i *desirable* for execution for a given user. The *desirable* fluent is added to the preconditions of each action. For example, a typical knowledge-producing action that senses the temperature of a room is defined in Table 3. Note that *desirable* conditions need to be specified either in the initial state, or (most likely) using DSURs. We should also separate user constraints from those used for search control. To that end

Action	Precondition	Effects
$sTemp(x)$	$K(room(x))$ $K(heatOn(x))$ $K(desSTemp(x))$	$add(K_v, temp(x))$

Table 3: $sTemp$ action.

we can introduce a fluent $indA_i$ that encodes the control information that makes action a_i *indicated*. These *indicated* fluents serve as an optimization mechanism that allows us to add further restrictions to the search space, in order to avoid unnecessary branching (e.g., it is *indicated* to sense the temperature of a room only if the agent already knows there is someone there). However, for simplicity, in this paper we express search control constraints using the *desirable* fluents.

We think that this form of action specification in PKS is well suited for WSC problems as it is easy to understand and maintain, as well as more extensible and reusable. Nevertheless, the knowledge engineer must be careful in introducing DSURs, because of the associated computational overhead.

The rest of this section describes a particular domain specification using PKS. Our actual experiments involve WSC for an air travel domain with five different variations on the problem of booking a flight between two cities under different customizing user constraints. For simplicity, we assume that all information regarding origin, destination, departure and arrival dates is already known and leave these parameters implicit. Otherwise, functional fluents can be introduced to model the knowledge acquisition. We envision that an executor agent can add all this complementary information at execution time. Moreover, there is significant advantage to this, both in terms of getting a more compact representation of the domain and increased performance at planning time.

All the problems described in this paper, share the basic domain specification elements given in Table 4. In this domain, four basic Web services are available: (i) $findRFlight(c)$, i.e., check if a flight exists on company c for the (implicit) desired cities and dates, (ii) $checkFSpace(c)$, i.e., check whether it has seats available, (iii) $checkFCost(c)$, i.e., find out its price and (iv) $bookFlight(c)$, i.e., book it. As expected, sensing actions correspond to information-gathering services and physical actions to world-altering services. We consider a fixed set of air companies and in the initial state, the agent always knows what these companies are (perhaps as a result of having used another existing information-gathering service). Additional action effects are represented by DSURs. In particular, rules (1) and (2) capture some effects of sensing for a flight. If the agent actually finds a flight, then it comes to know what the flight number is. On the other hand, if the agent finds that no such a flight exists, then it follows that the flight is not available. Rules (3) and (4) capture a simple search control constraint: the agent should only check for the price of a flight already known to have available space. The specifics of each version of the problem are captured by introducing additional DSURs.

Problem BPF: This problem involves a simple customiz-

Action	Precondition	Effects
$findRFlight(x)$	$K(airCo(x))$ $\neg K_w(flightExists(x))$ $K(desFindRFlight(x))$	$add(K_w, flightExists(x))$ $add(K_f, \neg desFindRFlight(x))$
$checkFSpace(x)$	$K(airCo(x))$ $\neg K_w(availFlight(x))$ $K_v(flightNum(x))$ $K(desCheckFSpace(x))$	$add(K_w, availFlight(x))$ $add(K_f, \neg desCheckFSpace(x))$
$checkFCost(x)$	$K(airCo(x))$ $\neg K_v(flightCost(x))$ $K(flightExists(x))$ $K(desCheckFCost(x))$	$add(K_v, flightCost(x))$ $add(K_f, \neg desCheckFCost(x))$
$bookFlight(x)$	$K(airCo(x))$ $\neg K(bookedFlight(x))$ $K(availFlight(x))$ $K(desBookFlight(x))$	$add(K_f, bookedFlight(x))$ $del(K_f, availFlight(x))$ $add(K_f, \neg desBookFlight(x))$
Domain specific update rules		

$K(airCo(x)) \wedge \neg K_v(flightNum(x)) \wedge K(flightExists(x)) \Rightarrow$
$add(K_v, flightNum(x))$ (1)

$K(airCo(x)) \wedge \neg K(\neg availFlight(x)) \wedge K(\neg flightExists(x)) \Rightarrow$
$add(K_f, \neg availFlight(x))$ (2)

$K(airCo(x)) \wedge \neg K_w(desCheckFCost(x)) \wedge K(availFlight(x)) \Rightarrow$
$add(K_f, desCheckFCost(x))$ (3)

$K(airCo(x)) \wedge \neg K_w(desCheckFCost(x)) \wedge K(\neg availFlight(x)) \Rightarrow$
$add(K_f, \neg desCheckFCost(x))$ (4)

Table 4: Basic air travel domain action specification.

ing user constraint: (s)he has a preferred company. In the initial state, the agent knows what the user's preferred company is. It also knows that actions $findRFlight(x)$ and $checkFSpace(x)$ are always *desirable*. The goal is either to book a flight with the preferred company, or with any other company if the preferred one is not available. For convenience, we introduce a couple of abbreviations, i.e., (5) and (6):

$$KnowNoFlightExists \doteq \forall_k(x)[K(airCo(x)) \Rightarrow K(\neg flightExists(x))] \quad (5)$$

$$KnowNoAvailFlight \doteq \exists_k(x)[K(airCo(x)) \wedge K(flightExists(x))] \wedge \forall_k(x)[(K(airCo(x)) \wedge K(flightExists(x))) \Rightarrow K(\neg availFlight(x))] \quad (6)$$

(5) is self-explanatory and (6) encodes the case where there is no seat available. Note that quantifiers are restricted to range over the set of known objects/constants in the domain.

The goal can then be represented as shown in (7):

$$\begin{aligned} &\% \ \textit{book pref. company} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\quad K(prefAirCo = x)] \quad | \\ &\% \ \textit{if pref. company not available book any other} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\quad K(prefAirCo \neq x) \wedge \\ &\quad K(flightExists(prefAirCo)) \wedge \\ &\quad K(\neg availFlight(prefAirCo))] \quad | \\ &\% \ \textit{if pref. company has no flight book any other} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\quad K(prefAirCo \neq x) \wedge \\ &\quad K(\neg flightExists(prefAirCo))] \quad | \\ &\% \ \textit{no flight booked} \\ &KnowNoAvailFlight \quad | \\ &KnowNoFlightExists \end{aligned} \quad (7)$$

Also note the use of '|' to denote (ordinary) disjunctions. We need to add a few update rules to the initial specification to make this work. Update rule (8) says that if it is known that a company has no space on its flight, then it is not *desirable* to book it. Update rule (9) says that a flight from a company other than the preferred company can only be *desirable* to

book if the agent already knows that the preferred company is not available. Note that the positive case of rule (8) should also be added, i.e., it is *desirable* to book the preferred company if available.

$$\begin{aligned} &K(airCo(x)) \wedge K(\neg availFlight(x)) \wedge \\ &\neg K_w(desBookFlight(x)) \wedge \\ &\quad \Rightarrow add(K_f, \neg desBookFlight(x)) \end{aligned} \tag{8}$$

$$\begin{aligned} &K(airCo(x)) \wedge K(availFlight(x)) \wedge \\ &K(prefAirCo \neq x) \wedge \\ &\neg K_w(desBookFlight(x)) \wedge \\ &K(\neg desBookFlight(prefAirCo)) \\ &\quad \Rightarrow add(K_f, desBookFlight(x)) \end{aligned} \tag{9}$$

Problem BMxF: This problem involves a different customizing user constraint, the user has a maximum price that (s)he is willing to pay. As before, the agent initially knows that actions $findRFlight(x)$ and $checkFSpace(x)$ are always *desirable*. Additionally, the agent knows what the user's price limit is. For convenience, we introduce another abbreviation, (10):

$$\begin{aligned} &KnowNoBudgetFlight \doteq \\ &\quad \exists_k(x)[K(airCo(x)) \wedge K(flightExists(x)) \wedge \\ &\qquad K(availFlight(x))] \wedge \\ &\quad \forall_k(x)[(K(airCo(x)) \wedge K(flightExists(x)) \wedge \\ &\qquad K(availFlight(x))) \\ &\qquad\quad \Rightarrow K(priceGtMax(x))] \end{aligned} \tag{10}$$

(10) represents the case where there is at least one available flight but none within the budget. The goal is to book any flight with price equal or less than the maximum price tag. This is represented by (11):

$$\begin{aligned} &\%\ \textit{book company within budget} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\qquad K(\neg priceGtMax(x))] \quad | \\ &\%\ \textit{no flight booked} \\ &KnowNoBudgetFlight \quad | \\ &KnowNoAvailFlight \quad | \\ &KnowNoFlightExists \end{aligned} \tag{11}$$

We must also add some DSURs to the initial specification. The update rule (12) captures the notion that if the cost of a flight is known, then it is also known whether or not it is greater than the maximum price. Note that in order to branch on the truth value of the inequality $(flightCost(x) > userMaxPrice)$ we need to introduce the fluent $priceGtMax(x)$ (the current implementation of PKS is unable to evaluate expressions that cannot be reduced to a numeric quantity at plan time; this issue is supposed to be addressed in a coming release of the planner). Rules (13) and (14) encode the maximum price constraint as only flights that do not exceed the maximum price tag should be considered for booking.

$$\begin{aligned} &K(airCo(x)) \wedge \neg K_w(priceGtMax(x)) \wedge \\ &K_v(userMaxPrice) \wedge K_v(flightCost(x)) \\ &\quad \Rightarrow add(K_w, priceGtMax(x)) \end{aligned} \tag{12}$$

$$\begin{aligned} &K(airCo(x)) \wedge K(\neg priceGtMax(x)) \wedge \\ &\neg K_w(desBookFlight(x)) \\ &\quad \Rightarrow add(K_f, desBookFlight(x)) \end{aligned} \tag{13}$$

$$\begin{aligned} &K(airCo(x)) \wedge K(priceGtMax(x)) \wedge \\ &\neg K_w(desBookFlight(x)) \wedge \\ &\quad \Rightarrow add(K_f, \neg desBookFlight(x)) \end{aligned} \tag{14}$$

Problem BPMxF: This problem considers not one but two customizing user constraints, preferred company and maximum price. In the initial state, the agent knows what the user's preferred company and maximum price are. In addition, the agent knows that actions $findRFlight(x)$ and $checkFSpace(x)$ are always *desirable*. The most important constraint is not exceeding the maximum price. Also, the user wants to book with the preferred company if possible. This goal is represented by (15):

$$\begin{aligned} &\%\ \textit{book pref. company if within budget} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\qquad K(prefAirCo = x) \wedge \\ &\qquad K(\neg priceGtMax(x))] \quad | \\ &\%\ \textit{if pref. company is above max. price} \\ &\%\ \textit{book another company within budget} \\ &\exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ &\qquad K(prefAirCo \neq x) \wedge \\ &\qquad K(\neg priceGtMax(x)) \wedge \\ &\qquad K(\neg desBookFlight(prefAirCo))] \quad | \\ &\%\ \textit{no flight booked} \\ &KnowNoBudgetFlight \quad | \\ &KnowNoAvailFlight \quad | \\ &KnowNoFlightExists \end{aligned} \tag{15}$$

This problem is a combination of BPF and BMxF and requires the addition of several DSURs to the original specification. In particular, update rules (12), (14) and (16) need to be added.

$$\begin{aligned} &K(airCo(x)) \wedge K(\neg priceGtMax(x)) \wedge \\ &K(prefAirCo \neq x) \wedge \\ &\neg K_w(desBookFlight(x)) \wedge \\ &K(priceGtMax(prefAirCo)) \wedge \\ &\quad \Rightarrow add(K_f, desBookFlight(x)) \end{aligned} \tag{16}$$

Note that (16) is a variation of (9) and (13) that encapsulates a stronger constraint: flights from a company other than the preferred one and not exceeding the maximum price should be considered for booking, provided the price for the preferred company exceeds the maximum price. Also note that for this to work, the positive case for the preferred company has to be added as well.

Problem BBF: This problem involves an optimization task: the user wants to book the best, i.e., cheapest flight available. Initially, the agent knows that actions $findRFlight(x)$ and $checkFSpace(x)$ are always *desirable*. It also knows that one of the air companies is the best/cheapest but it does not

know which one. The goal is to book the least expensive flight available, as shown in (17):

$$\begin{array}{l} \%\ \textit{book least expensive company} \\ \exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ \qquad K(bestAirCo = x)] \quad | \\ \%\ \textit{no flight booked} \\ KnowNoAvailFlight \quad | \\ KnowNoFlightExists \end{array} \tag{17}$$

The initial knowledge state includes a formula of the form $(bestAirCo = c_1|bestAirCo = c_2|...|bestAirCo = c_n)$ in K_x. Note the introduction of the new fluent $bestAirCo$, which represents the least expensive available company. Again we must add some update rules to the initial specification. The most important rules are presented below. The agent must figure out the ordering of the companies by flight price; this is captured by rules (18) and (19). Then we rely on the planner's ability to deal with disjunctive knowledge (exclusive *or*) of literals. Update rules (20) and (21) eliminate companies one by one based on flight price, until the agent can conclude that the last remaining company must be the best. Note that if two companies have equal price the agent arbitrarily picks one known to be available. Rule (22) eliminates companies that are not available. Once 'the best company' is found, rule (23) is triggered and we can proceed with the booking.

$$\begin{array}{l} K(airCo(x)) \wedge K(airCo(y)) \wedge \\ K(x \neq y) \wedge \neg K_w(priceEq(x,y)) \wedge \\ K_v(flightCost(x)) \wedge K_v(flightCost(y)) \wedge \\ \qquad \Rightarrow add(K_w, priceEq(x,y)) \end{array} \tag{18}$$

$$\begin{array}{l} K(airCo(x)) \wedge K(airCo(y)) \wedge K(x \neq y) \wedge \\ \neg K_w(priceLt(x,y)) \wedge K(\neg priceEq(x,y)) \wedge \\ \qquad \Rightarrow add(K_w, priceLt(x,y)) \end{array} \tag{19}$$

$$\begin{array}{l} K(airCo(x)) \wedge K(airCo(y)) \wedge \\ K(x \neq y) \wedge \neg K(bestAirCo \neq y) \wedge \\ K(priceLt(x,y)) \wedge K(availFlight(x)) \\ \qquad \Rightarrow add(K_f, bestAirCo \neq y) \end{array} \tag{20}$$

$$\begin{array}{l} K(airCo(x)) \wedge K(airCo(y)) \wedge \\ K(x \neq y) \wedge \neg K(bestAirCo \neq y) \wedge \\ K(priceEq(x,y)) \wedge K(availFlight(x)) \wedge \\ \qquad \Rightarrow add(K_f, bestAirCo \neq y) \end{array} \tag{21}$$

$$\begin{array}{l} K(airCo(x)) \wedge \neg K(bestAirCo \neq x) \wedge \\ K(\neg availFlight(x)) \\ \qquad \Rightarrow add(K_f, bestAirCo \neq x) \end{array} \tag{22}$$

$$\begin{array}{l} K(airCo(x)) \wedge \neg K(desBookFlight(x)) \wedge \\ K(bestAirCo = x) \\ \qquad \Rightarrow add(K_f, desBookFlight(x)) \end{array} \tag{23}$$

Problem BBPF: This problem is a refined version of BBF. The goal consists in booking the least expensive available flight, but if two flights have the same price the agent should always favour the preferred company. There are two user constraints, preferred company and least expensive flight. As before, in the initial state the agent knows that one of the air companies is the best (i.e., least expensive), but it does not know which one. Additionally, the agent knows what the preferred company is. The goal is represented by (24):

$$\begin{array}{l} \%\ \textit{book pref. company if least expensive} \\ \exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ \qquad K(prefAirCo = x) \wedge K(bestAirCo = x)] \quad | \\ \%\ \textit{if pref. company is not least expensive} \\ \%\ \textit{book the least expensive one} \\ \exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ \qquad K(prefAirCo \neq x) \wedge K(bestAirCo = x) \wedge \\ \qquad K(\neg desBookFlight(prefAirCo))] \quad | \\ \%\ \textit{no flight booked} \\ KnowNoAvailFlight \quad | \\ KnowNoFlightExists \end{array} \tag{24}$$

This preference is encoded by adding DSURs (18), (19), (20), (22), (23), and (25) to the original specification.

$$\begin{array}{l} K(airCo(x)) \wedge \neg K(bestAirCo \neq x) \wedge \\ K(prefAirCo \neq x) \wedge \\ K(priceEq(x, prefAirCo)) \wedge \\ K(availFlight(prefAirCo)) \\ \qquad \Rightarrow add(K_f, bestAirCo \neq x) \end{array} \tag{25}$$

For problems BMxF, BBF and BBPF, we think that an even more compact representation can still be achieved once the planner is enhanced with built-in support for conditional branching on the truth value of some unevaluated expressions (e.g., $f(x) < c$).

These examples are interesting because they encompass very general properties found in current WSC problems. In particular, they represent typical customizing user constraints, both hard constraints (true or false) and optimization constraints. In our approach, user's preferences are encoded using the *desirable* fluents and maintained by DSURs. However, as the number of constraints increases, adding DSURs can become tricky and also compromises the conciseness of the representation. This naturally raises the issue of exploring a systematic way of automatically generating DSURs from some given user preferences. We are currently investigating the feasibility of a such systematization. In general terms, we want to take as input a description of the user's preferences (i.e., in the form of a goal and some customizing constraints), and systematically generate a set of DSURs to enforce these constraints as the planner looks for a plan that achieves the goal.

Experimental Results

Our experiments were conducted on a 3.0GHz Xeon with 4Gb RAM, under Linux 2.4.22. The performance results are presented in Tables 5 and 6. The trials were performed by

# Co.	BPF	BMxF	BPMxF	BBF	BPBF
2	0.00	0.00	0.00	0.02	0.10
3	0.00	0.01	0.01	0.35	1.58
4	0.00	0.01	0.01	53.99	259.39
5	0.01	0.02	0.02	$> t_{max}$	$> t_{max}$
10	0.01	0.04	0.04	$> t_{max}$	$> t_{max}$
20	0.04	0.05	0.06	$> t_{max}$	$> t_{max}$
50	0.31	0.51	0.60	$> t_{max}$	$> t_{max}$
100	2.47	3.15	3.43	$> t_{max}$	$> t_{max}$

Table 5: Results for BPF, BMxF, BMxPF, BBF and BBPF with # Co. air companies using depth-first search with 1 explicit parameter, the company ($t_{max} = 300\ secs.$).

# Co.	BPF	BMxF	BPMxF	BBF	BPBF
2	0.17	0.21	0.37	1.27	8.42
3	0.58	0.72	1.20	24.02	109.33
4	1.45	2.20	3.71	$> t_{max}$	$> t_{max}$
5	3.76	4.33	4.65	$> t_{max}$	$> t_{max}$
10	80.60	96.45	105.49	$> t_{max}$	$> t_{max}$

Table 6: Results for BPF, BMxF, BMxPF, BBF and BBPF with # Co. air companies using depth-first search with 5 explicit parameters: company, origin, destination, departure date and arrival date ($t_{max} = 300\ secs.$).

running the planner[1] five times on each problem, and taking the average results for each one. All times are reported in CPU seconds.

In terms of performance, our approach scales up particularly well for the hard constraints problems BPF, BMxF and BPMxF. In particular, we can generate plans for 100 companies in less than 4 seconds. Note that, for the five parameters version of the problems, the running time increases by a factor of 100. This is why we eliminated the origin/destination cities, and departure/arrival dates parameters. Also note that the optimization problems BBF and BPBF appear not to scale up well. Both versions require a complete ordering of companies by cost, and therefore the planner has to do more complex combinatorial reasoning.[2]

Note that all our examples have at least twice as many DSURs as the most complex domains (i.e., medicate and opening a safe) reported on (Petrick & Bacchus 2002). Our experiments revealed that the overall performance is more sensitive to the number of parameters used than it is to the number of updates rules.

In terms of correctness, in all cases, the plans generated eventually succeed in booking a flight. Initially, most solutions contained many irrelevant action instances. First, we tried to improve on the quality of the plans by using breadth-first search instead of undirected depth-first search, but it does not scale up well. Therefore, all the results reported in

[1]The latest available release of PKS at the time of writing this paper was v0.6-alpha-2 (Linux)

[2]Also, the planner's current handling of the K_v database has not yet been optimized. It remains to be seen whether a scalable approach to this type of problems can be developed.

```
findRFlight(c1)
<branch, flightExists(c1)>
<k+>:
   checkFSpace(c1)
   <branch, availFlight(c1)>
   <k+>:
      checkFCost(c1)
      bookFlight(c1)
   <k->:
<k->:
   findRFlight(c2)
   <branch, flightExists(c2)>
   <k+>:
      checkFSpace(c2)
      <branch, availFlight(c2)>
      <k+>:
         checkFCost(c2)
         bookFlight(c2)
      <k->:
   <k->:
```

Table 7: Plan for problem BPF for two companies (generated using depth-first search).

Tables 5 and 6 were obtained using depth-first search. In the end, we managed to get rid of the unnecessary operators by carefully introducing appropriate search control constraints encoded into *desirable* fluents. We also intend to examine how iterative deepening search performs. A sample plan generated for the simplest problem appears in Table 7.

Agent-Based Infrastructure for WSC and Provisioning

Our work on planning for WSC is set within a broader effort to develop next generation tools for WSC and provisioning. We are mostly interested in Web services that require the use of agent-oriented approaches and planning techniques. In other words, we target our efforts to the new emerging semantic Web paradigm. To this end, we are currently working on an agent-oriented framework for inter-operable, intelligent multi-agent systems for WSC (Martínez 2004). Our approach is to interface three existing tools, IndiGolog, JADE, and PKS to obtain a toolkit that can provide an adequate infrastructure for doing semantic WSC and provisioning.

We use the IndiGolog agent programming language to address the need for reasoning, planning, execution monitoring and re-planning capabilities. IndiGolog (De Giacomo & Levesque 1999) is part of the Golog-family (De Giacomo, Lespérance, & Levesque 2000; Reiter 2001) of high-level logic programming languages developed by the Cognitive Robotics Group at the University of Toronto. IndiGolog provides a practical framework for implementing autonomous agents as it supports plan execution, sensing, exogenous events and planning in incompletely known dynamic environments. It also has mechanisms for execution monitoring and re-planning. The latter capabilities are important for Web service enactment, as the agent can keep track of how

responsive particular services are and control how much is delegated to them. However, IndiGolog is mainly intended for designing individual autonomous agents.

The Java Agent Development Framework (JADE) is a Java-based, FIPA-compliant software framework for developing multi-agent applications. JADE (Bellifemine, Poggi, & Rimassa 1999) provides interoperability with other agent applications/platforms that are compliant with the Foundation for Intelligent Physical Agents (FIPA) specifications. It also provides support for different types of ontologies. We use JADE as the front-end component to do matchmaking and negotiation of the services required. However, JADE's reasoning capabilities are very limited.

The PKS planner is included in the toolkit to provide a capability for generating conditional plans that include sensing actions. However, PKS's efficiency comes at the price of some limited representational and inferential power.

Each of these tools has its strengths and weaknesses, but combined together, they provide a very powerful toolkit. We have already developed an IndiGolog-JADE interface library, where the agents communicate using FIPA ACL (Agent Communication Language) messages. A JADE-PKS interface library is under development.

Conclusion and Future Work

In this paper, we addressed the problem of automated WSC by using knowledge-based planning techniques. In particular, we used the PKS planning system to synthesize plans that compose various available Web services to achieve user defined goals in a travel services domain. We studied how the running time of the planner depends on various problem parameters. We also described an approach to domain specification in PKS that is general, modular, customizable and intuitive. Despite the limited expressiveness and inferential power intrinsic to this approach, we provided empirical evidence that it is effective for representing/solving some typical WSC problems. Nevertheless, a great deal of the solution still depends on the knowledge engineer's design decisions, not the planner. This obviously raises the issue of looking for a systematic way to automatically generate a set of DSURs from a given description of the user's preferences, in the form of a goal and some customization constraints. Such a set of DSURs should enforce these constraints as the planner looks for a plan that achieves the goal. We also showed that this approach scales relatively well under different user constraints. We discussed as well a next generation toolkit and agent-oriented infrastructure for advanced WSC and provisioning.

In terms of scalability, one issue that arises from these experiments is the question of how sensitive our approach is with respect to the number of primitive actions. In particular, we should explore further the effects of unrelated actions or actions that do not get to execute, both with and without control information. Additionally, the problem of selecting which Web services are relevant for a particular WSC should be considered. In principle, we envision that some other part of our architecture takes care of that perhaps using some kind of heuristics. Finally, we should keep in mind that the current PKS planning system is still experimental, and future enhancements (e.g., more sophisticated search control mechanisms) should have a direct effect on the overall scalability.

Acknowledgements

We thank Ron Petrick from the Cognitive Robotics Group at the University of Toronto for providing an initial alpha version of the PKS planner as well as useful advice. Also, the referees provided some useful suggestions.

References

Bacchus, F., and Petrick, R. 1998. Modeling an agent's incomplete knowledge during planning and execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-1998)*, 432–443. San Francisco, CA: Morgan Kaufmann Publishers.

Bellifemine, F.; Poggi, A.; and Rimassa, G. 1999. Jade: A FIPA-compliant agent framework. In *Proceedings of PAAM-1999*, 97–108.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in non-deterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 473–478.

De Giacomo, G., and Levesque, H. J. 1999. An incremental interpreter for high-level programs with sensing. In Levesque, H. J., and Pirri, F., eds., *Logical Foundations for Cognitive Agents*. Springer-Verlag. 86–102.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Gottschalk, K., and IBM Team. 2000. Web services architecture overview: The next stage of evolution for e-business. Article, IBM, http://www-106.ibm.com/developerworks/web/library/w-ovr/.

Martínez, E. 2004. Web service composition as a planning task: an agent-oriented framework. MSc thesis, Department of Computer Science, York University, Forthcoming.

McIlraith, S., and Son, T. C. 2002. Adapting Golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR-2002)*, 482–493.

Petrick, R. P. A., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, 212–221. Menlo Park, CA: AAAI Press.

Petrick, R. P. A., and Bacchus, F. 2003. Reasoning with conditional plans in the presence of incomplete knowledge. In *Proceedings of the ICAPS-03 Workshop on Planning under Uncertainty and Incomplete Information*, 96–102. Trento, Italy: Università di Trento.

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