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, 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 $\alpha$ 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.
- $K_u$: This database contains information about function values. In particular, $K_u$ can store unasserted function terms whose values are known to the agent at execution time. $K_u$ is used for plan time modelling of the effect of sensing actions that return numeric values.
- $K_{\bar{w}}$: This database contains information about disjunctive (exclusive or) knowledge of ground literals of the form $(l_1\lor l_2\ldots\lor l_n)$. Intuitively, this formula represents the fact that the agent knows that 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 $K_f$, knowledge of whether an atom holds (or what a function value is), by putting it in $K_w$ (or in $K_u$); and knowledge that only one of a finite set of literals is true using $K_{\bar{w}}$. 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) $K_f(\alpha)$, is $\alpha$ known to be true? (ii) $K(\neg\alpha)$, is $\alpha$ known to be false? (iii) $K_w(\alpha)$, does the agent know whether $\alpha$? (iv) $K_{\bar{w}}(t)$, does the agent know the value of $t$? (v) the negation of any of the previous queries. In the above, $\alpha$ represents any ground atomic formula, and $t$ represents any variable free term. Complex queries include primitive queries, conjunctions of queries, disjunctions of queries, and quantified queries over the set of known objects. Complex goals can be expressed as complex queries.

**Actions:** Actions are specified in terms of three components: parameters, preconditions, and effects. For example in Table 1, the specifications for actions $\text{open}(r)$ and $\text{search}(r)$ 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.

<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{open}(r)$</td>
<td>$K(\text{room}(r))$ &amp; $K(\neg\text{opened}(r))$</td>
<td>$\text{add}(K_f, \text{opened}(r))$</td>
</tr>
<tr>
<td>$\text{search}(r)$</td>
<td>$\neg K_w(\text{found}(r))$ &amp; $K(\text{opened}(r))$</td>
<td>$\text{add}(K_w, \text{found}(r))$</td>
</tr>
</tbody>
</table>

Table 1: $\text{open}$ and $\text{search}$ actions.

**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 query 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 \( K(P(\bar{x}) \lor Q(\bar{x})) \). 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 instances of this in the BBF and BPBF problems below.

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 we also need to specify the values of desirable fluents in the initial state and now they change as actions are performed using DSURs. We should also separate user preference constraints from those used for search control. To that end we can introduce a fluent \( indA \), 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

<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>sTemp(x)</td>
<td>( K(room(x)) )</td>
<td>add(K, temp(x))</td>
</tr>
<tr>
<td></td>
<td>( K(heatOn(x)) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( K(desSTemp(x)) )</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: \( sTemp \) action.
of each version of the problem are captured by the goal, initial state specification, and a set of additional DSURs.

**Problem BPF:** This problem involves a simple user customization 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 \( \text{findRFlight}(x) \) and \( \text{checkFSpace}(x) \) 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):

\[
\begin{align*}
\text{KnowNoFlightExists} & \doteq \\
\forall_k(x)[K(\text{airCo}(x)) \Rightarrow K(\neg \text{flightExists}(x))] \\
\text{KnowNoAvailFlight} & \doteq \\
\exists_k(x)\{K(\text{airCo}(x)) \land K(\text{flightExists}(x))\} \land \\
\forall_k(x)[K(\text{airCo}(x)) \Rightarrow K(\neg \text{availFlight}(x))] \land \\
\forall_k(x)[(K(\text{airCo}(x)) \land K(\text{flightExists}(x))) \Rightarrow K(\neg \text{availFlight}(x))] \\
\end{align*}
\]

(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):

\[
\begin{align*}
\% \text{ book pref. company} \\
\exists_k(x)[K(\text{airCo}(x)) \land K(\text{bookedFlight}(x)) \land \\
K(\text{prefAirCo} = x)] \\
\% \text{ if pref. company not available book any other} \\
\exists_k(x)[K(\text{airCo}(x)) \land K(\text{bookedFlight}(x)) \land \\
K(\text{prefAirCo} \neq x) \land \\
(K(\neg \text{flightExists}(\text{prefAirCo})) \lor \\
K(\neg \text{availFlight}(\text{prefAirCo})))] \\
\% \text{ no flight booked} \\
\text{KnowNoAvailFlight} \lor \\
\text{KnowNoFlightExists}
\end{align*}
\]

Also note the use of ‘\(\lor\)’ 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 com-
company if available.

\[ K(\text{flightCost}(x)) \land K(\neg \text{availFlight}(x)) \land \\
\neg K_w(\text{desBookFlight}(x)) \land \\
\Rightarrow add(K_f, \neg \text{desBookFlight}(x)) \]

Problem BMxF: This problem involves a different user customization 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:

KnowNoBudgetFlight \( \equiv \)

\[ \exists_k(x)[K(\text{flightCost}(x)) \land K(\text{flightExists}(x)) \land \]
\[ \forall_k(x)[K(\text{flightCost}(x)) \land \\
K_w(\text{flightExists}(x)) \land K_w(\text{availFlight}(x))] \land \\
\forall_k(x)[K(\text{flightCost}(x)) \land \\
\neg K(\text{flightExists}(x))) \land \\
\Rightarrow K(\text{priceGtMax}(x))]
\]

(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 as follows:

% book company within budget
\[ \exists_k(x)[K(\text{flightCost}(x)) \land K(\text{bookedFlight}(x)) \land \\
K(\neg \text{priceGtMax}(x))]
\]

% no flight booked
KnowNoBudgetFlight | KnowNoAvailableFlight | KnowNoFlightExists

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 branch on the truth value of 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.

\[ K(\text{flightCost}(x)) \land \neg K_w(\text{priceGtMax}(x)) \land \\
K_w(\text{userMaxPrice}) \land K_w(\text{flightCost}(x)) \land \\
\Rightarrow add(K_w, \text{priceGtMax}(x)) \]

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 the preferred company if possible. For formal details of the goal see (Martínez & Lespérance 2004b).

Problem BBF: This problem involves an optimization task: the user wants to book the 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 (15):

% book least expensive company
\[ \exists_k(x)[K(\text{flightCost}(x)) \land K(\text{bookedFlight}(x)) \land \\
K(\text{bestAirCo} = x)] \land \\
% no flight booked
KnowNoAvailableFlight | KnowNoFlightExists

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 (16) and (17). Then we rely on the planner’s ability to deal with disjunctive knowledge (exclusive or) of literals. Update rules (18) and (19) 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 (20) eliminates companies that are not available. Once the best company is found, rule (21) is triggered and we can proceed
with the booking.

\[
\begin{align*}
K(airCo(x)) \land K(airCo(y)) \land \\
K(x \neq y) \land \neg K_w(priceEq(x, y)) \land \\
K_v(flightCost(x)) \land K_v(flightCost(y)) \land \\
\Rightarrow \quad add(K_w, priceEq(x, y))
\end{align*}
\]

(16)

\[
\begin{align*}
K(airCo(x)) \land K(airCo(y)) \land K(x \neq y) \land \\
\neg K_w(priceLt(x, y)) \land K(\neg priceEq(x, y)) \land \\
\Rightarrow \quad add(K_w, priceLt(x, y))
\end{align*}
\]

(17)

\[
\begin{align*}
K(airCo(x)) \land K(airCo(y)) \land \\
K(x \neq y) \land \neg K(bestAirCo \neq y) \land \\
K(priceLt(x, y)) \land K(\text{availFlight}(x)) \land \\
\Rightarrow \quad add(K_f, bestAirCo \neq y)
\end{align*}
\]

(18)

\[
\begin{align*}
K(airCo(x)) \land \neg K(bestAirCo \neq y) \land \\
K(priceEq(x, y)) \land K(\text{availFlight}(x)) \land \\
\Rightarrow \quad add(K_f, bestAirCo \neq y)
\end{align*}
\]

(19)

\[
\begin{align*}
K(airCo(x)) \land \neg K(bestAirCo \neq x) \land \\
K(\neg \text{availFlight}(x)) \land \\
\Rightarrow \quad add(K_f, \text{bestAirCo} \neq x)
\end{align*}
\]

(20)

\[
\begin{align*}
K(airCo(x)) \land \neg K(desBookFlight(x)) \land \\
K(bestAirCo = x) \land \\
\Rightarrow \quad add(K_f, desBookFlight(x))
\end{align*}
\]

(21)

**Problems BPF:** 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. For formal details of the goal see (Martínez & Lespérance 2004b).

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 user customization 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 developing a systematic way of automatically generating DSURs from some given user preferences. We are currently investigating the feasibility of 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 user customization 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 running the planner\(^1\) five times on each problem, and taking the average results for each one. All times are reported in CPU seconds.

<table>
<thead>
<tr>
<th># Co.</th>
<th>BPF</th>
<th>BMxF</th>
<th>BPMxF</th>
<th>BBF</th>
<th>BPBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>0.01</td>
<td>0.02</td>
<td>0.03</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td>5</td>
<td>0.03</td>
<td>0.07</td>
<td>0.10</td>
<td>41.70</td>
<td>43.01</td>
</tr>
<tr>
<td>10</td>
<td>0.97</td>
<td>36.25</td>
<td>46.20</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
</tr>
<tr>
<td>15</td>
<td>58.61</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
</tr>
</tbody>
</table>

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\(_{\text{max}}\) = 300 secs.).

<table>
<thead>
<tr>
<th># Co.</th>
<th>BPF</th>
<th>BMxF</th>
<th>BPMxF</th>
<th>BBF</th>
<th>BPBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.09</td>
<td>1.85</td>
<td>3.37</td>
<td>18.71</td>
<td>20.74</td>
</tr>
<tr>
<td>3</td>
<td>6.82</td>
<td>14.10</td>
<td>16.26</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
</tr>
<tr>
<td>4</td>
<td>20.95</td>
<td>106.75</td>
<td>139.62</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
</tr>
<tr>
<td>5</td>
<td>112.36</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
<td>&gt; t(_{\text{max}})</td>
</tr>
</tbody>
</table>

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\(_{\text{max}}\) = 300 secs.).

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 up to 10 companies in less than 60 seconds. Note that, for the five parameters version of the problems, the running time increases by a factor of 1000, here it checks all instances of conditions in DSURs at every step. This is why we eliminated the origin/destination cities, and departure/arrival dates parameters. However, for problems BPF, BMxF and BPMxF, PKS can still generate plans for 4 companies in less than 3 minutes. Also note that the optimization problems BBF and BBPF 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.\(^2\) We also tested how sensitive our approach is with respect to the number of primitive actions. This experiment consisted in adding several primitive actions representing irrelevant WSs, with some of them getting to execute and some

\(^1\)The release of PKS used was v0.7-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(c2)
<branch-1, flightExists(c2)>
<k+>:
    checkFSpace(c2)
    <branch-4, availFlight(c2)>
    <k+>:
        checkFCost(c2)
        bookFlight(c2)
    <k->:
        findRFlight(c1)
        <branch-5, flightExists(c1)>
        <k+>:
            checkFSpace(c1)
            <branch-6, availFlight(c1)>
            <k+>:
                checkFCost(c1)
                bookFlight(c1)
            <k->:
        <k->:
            findRFlight(c1)
            <branch-2, flightExists(c1)>
            <k+>:
                checkFSpace(c1)
                <branch-3, availFlight(c1)>
                <k+>:
                    checkFCost(c1)
                    bookFlight(c1)
            <k->: 
    <k->:
</k->

Table 7: Plan for problem BPF for two companies, with preferred company c2 (generated using depth-first search).

of them not. In general, the running time is less sensitive to the number of action preconditions than it is to the number of irrelevant primitive actions.

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 in (Petrick & Bacchus 2002). Our experiments reveal 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 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 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 fluent. 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 2005). Our approach is to interface three existing tools, IndiGolog (De Giacomo & Levesque 1999), JADE (Bellifemine, Poggi, & Rimassa 1999), 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 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 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. A high-level view of the main components of the resulting IG-JADE-PKSl Legislature depicted in Figure 1. 5 We obtained this toolkit by first developing an IndiGolog-JADE interface library, where the agents communicate using FIPA ACL (Agent Communication Language) messages. The motivation for this is to enhance IndiGolog agents’ capabilities to participate in multi-agent systems (MAS). As well, we want to provide a tool for building reasoning agents that is JADE compatible. Thus, our IndiGolog-JADE (IG-JADE) interface adds value to both tools by combining their

5All our contributions are shown in dotted lines.
 strengths. This naturally raises the issue of how to design, implement and deploy MAS involving heterogeneous platforms, i.e., rule-based and object-oriented.

One important technical issue that has to be addressed is the communication scheme of the participating agents/platforms. At a lower level of abstraction this involves routing the communication flows between platforms, and establishing what kind of communication protocols are supported by each platform. Our current implementation at a low-level uses TCP/IP sockets. At a higher level, our interface should also address other issues like the structure of the messages, the use of standard communications languages (e.g., FIPA ACL) vs. application specific languages, and similarly with respect to the use of content languages (e.g., FIPA SL). Note that, we extended the IndiGolog side by providing a direct interface to JADE, as well as adding library procedures to support FIPA agent interaction protocols.

In addition, we have also developed a JADE-PKS interface library (Martínez & Lespérance 2004a). We extended the JADE platform by providing direct interfaces to IndiGolog and to the PKS planning system respectively.

By using the IG-JADE-PKSlib, IndiGolog agents can not only communicate/interact with JADE agents using FIPA ACL (or any application specific language), but also with any FIPA-compliant agent platform. As well, JADE agents can exploit the reasoning and planning capabilities provided by IndiGolog and PKS.

We also developed the Air Travel Assistance (ATA), a Web service (WS) application prototype implemented using the IG-JADE-PKSlib toolkit. The ATA application is delivered as an inter-operable and heterogeneous multi-agent system (MAS), which can handle the same range of problems discussed earlier. The ATA application has a simple plan executor agent implemented in IndiGolog.

**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 at least for non-optimization constraints. We discussed as well a next generation toolkit and agent-oriented infrastructure for advanced WSC and provisioning.

In terms of scalability, obviously the main issue that arises from these experiments is the question of whether a more scalable approach to the optimization constraints type of problems can be developed. We are currently investigating more sophisticated search control mechanisms, but a more definite answer to this question should only come once the PKS implementation is enhanced with the features described in (Petrick & Bacchus 2004).

Given the effects of irrelevant actions/WSs on performance, another problem that requires more work is how to select which Web services are relevant for a particular WSC. We intend to investigate this further. In principle, we envision that in a multi-agent framework, a dedicated reasoning agent should take care of selecting relevant WS/actions perhaps using some kind of heuristics. This is a still open problem, currently being the subject of a great deal of research from the Semantic Web and Web Services communities (Srivastava & Koehler 2003).

Our approach so far has been pragmatic, focusing on what functionalities are more convenient for an application designer. One important issue that arises from this work is the question of how to provide mechanisms to translate standard ontology description languages used in WS applications (e.g., OWL (Bechhofer et al. 2004)) into the type of action theories and process specifications that IndiGolog and PKS use. Clearly, more work is required to understand all the requirements which should be supported by such a library.

Another limitation to our approach is that our conditional plans must handle all the possible results of sensing actions. This is made worse by the fact that we are dealing with a very large search space (the Web!) possibly populated with many matching Web services. However we can minimize to some extent these limitations by introducing some (heuristic-based) service discovery/filtering mech-
anisms prior to the planning stage. This should hopefully reduce the size of the search space used in planning. In addition, we can also reduce the level of uncertainty in the initial state, by performing some of the sensing before the start of the planning phase. In practice, one will likely need an incremental mechanism for planning, plan execution, and sensing. How to effectively interleave these activities remains to be investigated. We also want to enhance the executor to do execution monitoring and failure recovery.

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. Ksenia Choubina and Stan Zhiyan Li have made significant contributions to the development of the toolkit.

References


1.3. planning methods

The planning method is a method for conducting planned calculations, an order and an algorithm for justifying the plan's. Using the program-target method (complex target programs as a result of using this method), one can solve the most important problems of the development of the planning object, if they can not be solved by traditional means of planning and management. The application of the program-target planning method follows from the principle of priority direction in the economy. It is used to determine the demand for certain services or goods; changes in prices for raw materials, services of third-party organizations, when predicting the competitive situation in the market, etc. Motivated by the problem of automated Web service composition (WSC), we present some empirical evidence to validate the effectiveness of using knowledge-based methods. You can change your ad preferences anytime.


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 ... If a new Web service becomes available we can add it as a new primitive action to the domain specification.