A POP-Based Replanning Agent for Automatic Web Service Composition

Joachim Peer

MCM Institute, University of St. Gallen, Switzerland
joachim.peer@unisg.ch

Abstract. This paper illustrates how a modified version of a modern Partial Order Planner (POP) can be combined with a replanning algorithm to solve planning problems in Web service domains. The contributions of the work are (i) a method of using feedback gained from plan execution for improving plan search and (ii) a novel approach of dealing with nondeterministic Web service operations.

1 Introduction

Web services are distributed software components that can be exposed and invoked over the internet. Commonly, interface description languages such as the Web Service Description Language WSDL [1] are used to describe the syntactical interface of a Web service and the message streams it uses to communicate with its clients. In addition to that, semantic annotations can be created in order to provide software agents with information to reason about the capabilities and consequences of service operations.

In an environment of semantically annotated services, a user who needs to achieve certain goals can be assisted by software agents which automatically identify, invoke, compose and monitor services in order to accomplish the user's goals, which may be either explicitly stated or derived from the user's current situation.

Recently, several papers have investigated the potentials and boundaries of applying *AI planning techniques* to derive Web service processes that achieve the desired goals. It has been pointed out that automatic Web service composition (WSC) differs from classical planning domains in several aspects [2, 3].

First, there is the problem of *incomplete information*: Many planning problems in the Web service domain require the querying of information-providing services, for instance to check the availability of some product at an online retailer. Therefore, the classical planning assumption of a complete initial state description can not be maintained in the WSC domain. Further, there is the problem of *nondeterministic behavior* of services: Web service operations may fail during execution time or yield unexpected results, for instance when a retailer suddenly runs out of stock of some product.

To be practically useful, agents have to be able to handle these contingencies. In this paper, we describe how a modified Partial Order Planning (POP)

A. Gómez-Pérez and J. Euzenat (Eds.): ESWC 2005, LNCS 3532, pp. 47–61, 2005.

algorithm embedded in an execution monitoring agent can deal with the issues mentioned above.

The remainder of this paper is structured as follows: In Section 2, we will briefly describe the conceptual representation schema for Web services our work is based on and in Section 3, we describe the semantics of service executions. In Section 4, we present the structure of the planning problems our system can handle. Then, in Section 5, we lay out our proposed solution and present our contributions to the problem solving algorithms involved. In Section 6, we present examples to illustrate the system and we present our preliminary empirical tests. In Section 7, related work is discussed, followed by a summary of the contributions of the paper in Section 8.

2 Representing Web Service Semantics

Currently, there exist several proposals for markup formats for semantic Web services. Among them are OWL-S [4], WSMO/WSML [5] and the recent SESMA [6] markup format. For the present work we chose SESMA, because it is easy to use and provides convenient support to mark up nondeterministic service operations.

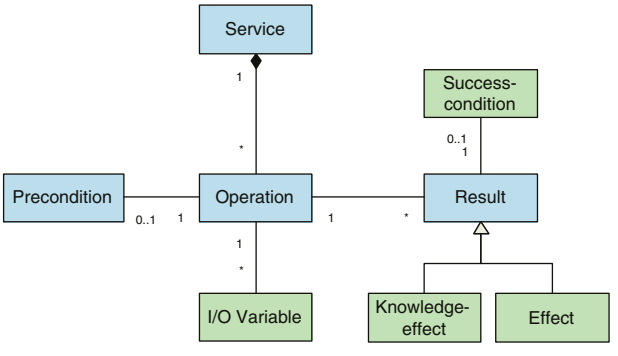


Fig. 1. Service representation in the SESMA model [6]

The syntax and semantics of the SESMA constructs are described in detail in [6]; here we give a brief overview: In the SESMA model, each service consists of a set of operations. Each operation can be written as a quadruple $op = \langle URI, Prec, Res, Var \rangle$. *URI(op)* denotes the unique identifier of the operation. *Prec(op)* denotes the optional precondition formula of the operation, which describes conditions that need to be satisfied when the operation is invoked. *Res(op)* denotes the set of possible results of the operation, which can be *effects* and *knowledge-effects*; both describe conditions that will hold after service execution, but while the world state as a whole may be changed to achieve an *effect*, the world state remains unchanged to achieve *knowledge-effects*; only

```
<op-def name="buyItem" wsdl:portType="ShopA">
  <input>
    <var name="?item" wsdl:part="ean"/>
    <var name="?cc" wsdl:part="ccNr" />
    <var name="?ccexp" wsdl:part="ccExpDate" />
  </input>
  <precondition>
    <and>
      <s:have-creditcard owner="$client"
         nr="?cc" expires="?ccexp"/>
      <s:in-catalog vendor="shopA.com" item="?item"/>
    </and>
  </precondition>
  <output>
    <var name="?result" wsdl:part="buyItemReturn"/>
  </output>
  <effect>
    <!-- to be checked after invocation -->
    <success-condition lang="java">
    !("no".equals(output.get("?result")))
   </success-condition>
    <!-- the desired effect -->
    <s:possess owner="$client" item="?item"/>
  </effect>
 </op-def>
```

```
<op-def name="getItemList" wsdl:portType="ShopA">
  <output>
    <var name="?item"
     wsdl:part="getItemListReturn"
     wsdl:path="Item/ean"/>
    <var name="?title"
     wsdl:part="getItemListReturn"
     wsdl:path="Item/title" />
    <var name="?desc"
     wsdl:part="getItemListReturn"
     wsdl:path="Item/description"/>
  </output>
  <knowledge-effect>
    <forall>
      <var name="?item" />
      <and>
        <s:in-catalog vendor="shopA.com"
           item="?item" />
        <s:has-title item="?item" title="?title" />
        <s:has-description item="?item"
           description="?desc" />
      </and>
    </forall>
  </knowledge-effect>
</op-def>
```

Fig. 2. SESMA markup for two operations

the *world view* of the agent may be affected by knowledge-effects. Since services often behave nondeterministically, each result $r \in Res(op)$ may have a success condition $SC(r)$, which can be used after service invocation to determine whether or not the invocation did achieve the desired effects.

Further, the set *Var(op)* defines the variables used in the formulas, whereby input variables are distinguished from output variables. The variable definitions tell the agent how the values represented by the variables are syntactically encoded in the input and output streams sent to/received from the service.

To illustrate the constructs sketched above, we depict the SESMA descriptions of two operations of a Web service `ShopA.com` in Fig. 2:

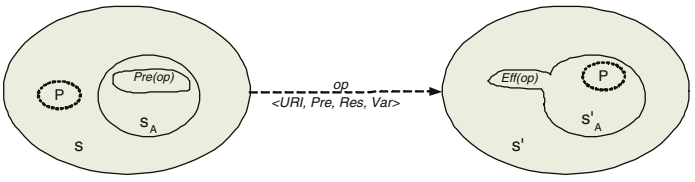
The operation on the left, `buyItem`, has a `<precondition>` that requires that the client has a credit-card ?*cc* and that the item ?*item* to be purchased is in the catalog of the shop. The operation has a single `<effect>`, which is that the client will possess the specified product, if the effect's `<success-condition>`, which may be written as a Java(TM) snippet, evaluates to *true* after service execution. The second operation, `getItemList`, has no precondition, but it has a single `<knowledge-effect>` formula. The knowledge effect specifies that, for all ?*item*-objects received from the service, there is an ?*item* in the catalog with a title ?*title* and a description ?*desc*.

The `<input>` and `<output>` sections of the operations specify the WSDL message parts and XML paths that identify the location of the variable values in the input and output messages.

3 Semantics of Service Executions

3.1 Representing the World and Its Dynamics

We assume that the world is represented in terms of a state description. Formally, we define a state s as a conjunction of positive atomic formulas (atoms) $L_1 \wedge$



(a) World prior to execution of *op*

(b) World after the execution of *op*

Fig. 3. The world and its dynamics are represented by states and state transitions

$... \wedge L_n$. Consequently, for each $L_i \in s$ holds $s \models L_i$ and for each $L_i \notin s$ holds $s \not\models L_i$.

As illustrated by Fig. 3, we distinguish between the global state s of a world and the state knowledge s_A of an agent A who reasons about this world. We assume that $s_A \subseteq s$. This means that an agent may have incomplete knowledge, but we do not assume that it has wrong knowledge.

The dynamics of the world are formalized as a state transition system, which can be thought of as a labeled directed graph: The vertices of the graph are the states, and the edges of the graph are the transitions that may take place, labeled by the operations which cause the state transition. Fig. 3 illustrates a transition from state s to a state s', triggered by an operation *op*. As shown in the illustration, the precondition *Pre(op)* is satisfied by s (and s_A), which allows the agent to invoke the operation. The operation has an effect *Eff(op)* which is added to s'_A. The operation also has a knowledge-effect; the consequence of the knowledge effect is that the fact P, which the agent was unaware of in its state s_A is added to s'_A, i.e. the agent becomes aware of P after the service execution.

3.2 Conversation Data Sets

Conversation data sets capture the data tokens that are exchanged between clients and services at runtime. Conversation data sets are an important prerequisite for evaluating SESMA formulas.

A conversation data set is a set of substitutions $\{\theta_1, ...\theta_n\}$. We denote a substitution as a finite set of the form $\theta = \{x_1/t_1, ..., x_n/t_n\}$, where each x_i is a distinct variable and each t_i is a constant, such that $x_i \neq t_i$, for all $1 \leq i \leq n$. Note that the variables of an operation are defined in the set $Var(op)$ (cf. Sect. 2). A substituted formula $F\theta$ is a variant of formula F where all variables x in F are replaced by a constant c if there is an element $\{x/c\} \in \theta$.

Let us consider the role of conversation data sets and substitutions in service execution:

- Substituting *input variables*: When invoking an operation *op*, the agent specifies the values certain input variables should have. This specification is represented by a conversation data set I and each substitution in I contains at most one term t_i for every variable x_i of the input variables defined for the operation.

– Substituting *output variables*: Analogously, the values returned by the service are captured by a conversation data set O, which contains an arbitrary number of substitutions ϑ, where each substitution contains at most one term t_o for every variable x_o of the output variables defined for the operation.

Let us illustrate this using the example markup of Fig. 2: For a particular invocation of the operation `getPrice`, the user might define the input data set $I = \{\{?item/44300\}\}$, and the service might return an output data set $O = \{\{?price/79.00\}\}$. The operation `getItemList` may return an output data set $O = \{ \{?item/44300, ?title/CAM300, ?desc/WebCam\}, \{?item/44340, ?title/HS340, ?desc/HeadSet\} \}$, which contains *multiple* substitutions, representing catalog data of an online retailer service.

Next, we define how conversation data sets can be *combined*. We need combined conversation data sets to materialize effects (cf. Sect. 3.5). We start by defining the combination of substitutions: Given two substitutions $\theta = \{x_1/t_1, ..., x_k/t_k\}$ and $\vartheta = \{x_m/t_m, ..., x_n/t_n\}$, we define the joint substitution $\theta + \vartheta = \{x_1/t_1, ..., x_k/t_k, x_m/t_m, ..., x_n/t_n\}$. Given an input data set I and an output data set O, we define the *combined conversation data set* $C = I \otimes O$ as follows: For each $\theta \in I$ and each $\vartheta \in O$, C contains an element $\{\theta + \vartheta\}$.

3.3 Evaluating Preconditions

The precondition is evaluated against the state s, whereby the values in the conversation data set I provide the substitutions for the variables in the precondition. A precondition P is satisfied iff $s \models P\theta$ holds, for every substitution θ in the input data set I.

Since the agent is only aware of the subset s_A of s it must test the precondition against s_A. As long as the agent can calculate the truth value of the precondition using the literals in s_A, it can conclude that the precondition is satisfied by s as well. However, if the agent has to resort to closed world assumption, i.e. if it assumes $s_A \not\models L^+$ for some positive literal $L^+ \notin s_A$, then there is the possibility that the evaluation against s_A differs from the evaluation against s.

3.4 Evaluating Success Conditions

As mentioned in Section 2, SESMA supports the markup of nondeterministic service results. For instance, a payment Web service may send a note either confirming the transaction or reporting a failure. When dealing with such nondeterministic operations, the agent should only assert the effect of the operation after it has executed it *and* after it received a confirmation of success. For instance, the markup of the operation *buyItem* listed in Fig. 2 defines a success condition which depends on the value of the output variable *?result*.

SESMA success-conditions, which can be written either as declarative SESMA formulas or as boolean Java(TM) expressions, contain the information needed to determine the success or failure of an operation's result(s). We formalize the evaluation as a function $eval(SC(r), s_A, I, O) \mapsto \{true, false\}$. The inputs to

this function are the success condition SC of the result r whose success is to be determined, the known pre-execution state s_A , the input data set I and the output data set O. The output of the evaluation function is a Boolean value, reflecting whether or not the result r did indeed occur and if the effect formula can be materialized. If a result has no success condition, then *eval* will return *true* as a default. Note that it is permitted that one effect of an operation occurs while another effect of the same operation does not.

3.5 Materializing Effects

If an operation is executed without errors, then the successor state can be calculated. All results of the operation, whose success conditions evaluate to *true* or which lack a success condition, are considered; those results whose success conditions evaluate to *false* are ignored.

A result formula E is *materialized* in a successor state s' iff $s' \models E\vartheta$ holds for every substitution ϑ in the combined data set $I \otimes O$.

To formally describe the truth value of an atom A in a given state we introduce the function *val*, which defines for each atom and for each state whether or not the atom is true in that state. Given a state transition from state s to s' by invocation of an operation *op*, the function is defined as follows:

$$val(s', A) = \begin{cases} true, & \text{if } s \models Pre(op)\theta \text{ for every } \theta \in I \\ & \text{and if there is a result } r \in Res(op) \\ & \text{where } eval(SC(r), s_A, I, O) = true \\ & \text{and } \exists \vartheta \in (I \otimes O) \text{ such that } r\vartheta \models A \\ false, & \text{if } s \models Pre(op)\theta \text{ for every } \theta \in I \\ & \text{and if there is a result } r \in Res(op) \\ & \text{where } eval(SC(r), s_A, I, O) = true \\ & \text{and } \exists \vartheta \in (I \otimes O) \text{ such that } r\vartheta \models \neg A \\ val(s, A), & \text{elsewhere.} \end{cases}$$

The first case states that A is true in the new state s' if (i) the precondition formula (whose variables are substituted by the input values) is satisfied by the pre-execution state and if (ii) A is a consequence of a result formula r (whose variables are substituted by the input and output values) and if (iii) the function *eval* returns *true* for the result. Analogously, the second case describes that A is not true in the new state, if $\neg P$ is a consequence of a result of the operation. The third case describes the situation where A is not affected by the execution of the operation, which addresses the frame problem of logical action descriptions.

From the agent's perspective, a *knowledge effect* is treated (and materialized) just like an *effect*. Nevertheless, there is a difference, which is worth some consideration: A knowledge effect KE of an operation describes an effect without side effects, i.e. it does not contribute to any changes occurring between s and s', but it may alter the world view s_A of the agent: the statement $[(s' \models KE\vartheta) \Rightarrow (s'_A \models KE\vartheta)] \wedge [(s' \not\models KE\vartheta) \Rightarrow (s'_A \not\models KE\vartheta)]$ holds, for every $\vartheta \in (I \otimes O)$.

4 Planning Goals

Having set out the world the semantic Web service agents act in, we characterize the structure and nature of the goals they are supposed to achieve: We view goal descriptions as logical statements about the desired world state. We allow the specification of goals in the form of literals (positive or negative atomic formulas) and conjunctions and disjunctions of literals, whereby variables are treated with existential quantification. For instance, the following SESMA construct is a valid goal; it is a conjunction of non-ground literals and can be fulfilled by purchasing an *?item* that has a product id (EAN number) "1234":

```
<and>
  <possess who="client" what="?item" />
  <has-ean item="?item" ean-nr="1234"/>
</and>
```

Further, we support the distinction between goal formulas that describe *achievement* goals and goal formulas that describe *information gathering* goals. Achievement and information gathering goals may be combined to form more complex goal descriptions.

To distinguish information gathering goals from achievement goals, we introduce the goal annotation *find-out*. A plan satisfying a goal formula G that is annotated by *find-out* may not contain an operation that has an *effect* G', if G and G' unify. Instead, plans that achieve the goal using *knowledge-effects* are sought in this case. This concept is derived from the idea of *maintenance goals* discussed in [7].

As an example, a goal `<find-out> <n:color item='?itm' value='?c'/> </find-out>` may only use operations that do *not* actively affect the color of item *?itm*; we would not want the agent to call an operation that sets the color to some new value and then reports that newly assigned color. Instead, the value should be gathered through a *knowledge effect* of an operation.

Goal expressions that are *not* annotated are interpreted as achievement goals, with the usual semantics, i.e. the agent can do whatever is needed to make the annotated formula true in s'. For instance, the goal `<n:possess item='someID'/>` means that the agent can change the world state as necessary to achieve the goal.

5 The Proposed Replanning Agent

In the following we will describe the architecture and algorithms we propose to solve problems in the world formalized as discussed in the last sections.

To deal with the nondeterministic nature of the domain and the arising contingencies, we chose an execution monitoring architecture, where a classical planner – an extension of VHPOP [8] (Versatile Heuristic Partial Order Planner) – is *embedded* in an execution monitoring engine. In Sect. 5.1 we briefly sketch VHPOP and the extension we added to it to fit into the replanning architecture; in Sect. 5.2 we then describe the replanning algorithm itself.

5.1 The POP-Based Kernel

VHPOP uses Partial Order Planning (POP), a well known planning technique where the reasoner searches a space of *plans*. A partially ordered plan is represented as a quadruple $\langle \mathcal{S}, \mathcal{O}, \mathcal{B}, \mathcal{L} \rangle$, which consists of the following components: $\mathcal{S}$ is a set of plan steps, i.e. instances of operations. $\mathcal{O}$ is a set of ordering constraints. Each ordering constraint is of the form $s_i \prec s_j$, which means that the step s_i must be executed *before* step s_j. If the set $\mathcal{S}$ of some plan π has at least two steps s_a and s_b where $\mathcal{O}$ neither contains $s_a \prec s_b$ nor $s_b \prec s_a$, then π is a *partially ordered* plan. $\mathcal{B}$ is a set of variable binding constraints on the parameters of action instances: Each variable constraint is of the form $var = x$ or $var \neq x$, where var is a variable of some plan step and x is either a constant value or a reference to a variable of some other plan step. If only ground plan steps are used, then $\mathcal{B} = \emptyset$. Finally, $\mathcal{L}$ is the set of causal links. Causal links are used to keep track why a step was introduced to a plan and to prevent other steps from interfering with that purpose. If a step s_i achieves a proposition p to satisfy a precondition of step s_j, the causal link $s_i \xrightarrow{p} s_j$ is added to $\mathcal{L}$.

Further, the following two derived sets are considered in partial order planning: (1) $\mathcal{OC}$ is the set of open conditions of a plan. An open condition $\xrightarrow{p} s$ emerges when p is a literal that is part of the precondition *Prec(s)* and when there is no causal link $s_x \xrightarrow{p} s$ in $\mathcal{L}$. In other words, open conditions are preconditions of plan steps which have not yet been addressed by the current plan. (2) $\mathcal{UL}$ is the set of unsafe links. A causal link $s_i \xrightarrow{p} s_j$ is called *unsafe* if there exists a step $s_k \in \mathcal{S}$ such that (i) $\neg p \in$*Eff*(s_k) and (ii) $\mathcal{O}$ is consistent with $\{s_i \prec s_k \prec s_j\}$. In such a case, s_k is said to *threaten* the causal link $s_i \xrightarrow{p} s_j$. The union of a plan's open conditions and unsafe links is called the set $\mathcal{F}$ of *flaws* of π, i.e. $\mathcal{F}(\pi) = \mathcal{OC}(\pi) \cup \mathcal{UL}(\pi)$. A plan π that has no flaws is called *complete*.

An open condition $\xrightarrow{p} s$ can be resolved by introducing or reusing a plan step that has an effect achieving p. On the other hand, a *threat* of a causal link $s_i \xrightarrow{p} s_j$ by a step s_k can be possibly resolved either by *demotion*, i.e. by adding an ordering constraint $s_k \prec s_i$ to $\mathcal{O}$ or by *promotion*, i.e. by adding $s_j \prec s_k$ to $\mathcal{O}$. If the planner uses lifted actions, i.e. if it allows action instances with variables in their parameter lists, a threat can also possibly be resolved by *separation*, that is by adding binding constraints such that p and $\neg p$ can not be unified.

The way a planner navigates through plan space, i.e. the strategy it employs to chose the plans to refine and the flaws to remove, determines the efficiency of the planner. VHPOP utilizes a planning graph and several search heuristics to steer the planning process into a fruitful direction; a discussion of these techniques can be found in [8].

We have extended VHPOP by introducing a set $\mathcal{AL}$ (which stands for "Avoid-Links"), which is provided as an additional argument to the main procedure Make-Plan and is subsequently handed over to Refinements and its various sub-routines, which are described in [8].

$\mathcal{AL}$ is a set of *causal link patterns* that must be avoided by the planner. This means that none of the partial plans $\pi \in \mathcal{P}$ devised by the planner may contain a causal link that matches a causal link pattern in $\mathcal{AL}$.

A causal link pattern is a triple $\langle x, p, y \rangle$ where p is a literal, x is either an operation *op* which has p in one of its results $Res(op)$ or x is a wildcard (asterisk, $*$), and where y is either an operation *op* whose precondition is (partially) fulfilled by x (i.e. $p \in Prec(op)$), or a wildcard ($*$). A causal link $s_i \xrightarrow{p} s_j$ *matches* a causal link pattern $x \xrightarrow{q} y$ iff

1. p unifies with q, and
2. $x = opr(s_i)$ or $x = *$, and
3. $y = opr(s_j)$ or $y = *$,

whereby *opr(s)* denotes the Web service operation represented by the plan step s. The reasons for maintaining the set $\mathcal{AL}$ of causal link restrictions are (i) to avoid plan structures that already failed in prior executions and (ii) to avoid plan structures that violate the restrictions imposed by *find-out* goals. We will discuss both issues in greater detail below.

5.2 The Replanning Algorithm

In the following we illustrate how to solve planning problems as described in Section 4 by breaking them down into a series of simpler planning problems to be tackled by the modified VHPOP algorithm described in the last section.

The main routine of the replanning agent is Solve-Problem (cf. Fig. 4); its arguments are the goal $\mathcal{G}$, the initial situation $\mathcal{I}$, a set *anno* of SESMA-annotations of the available Web services, and the number *threshold* of planning-and-execution cycles to be performed before giving up.

First, the Web service domain and the WSC problem given have to be translated into a PDDL based planning domain description $\mathcal{D}$ to be passed to the VHPOP algorithm (which uses PDDL as input format). This is done by calling the operation Build-Domain (cf. line 4). The transformation is straightforward: (i) For each annotated Web service operation a PDDL planning operator is created, whereby the SESMA precondition is transformed into a precondition of the operator. The effect of each planning operator is formed by a conjunction of all results of the corresponding operation, whereby the success conditions are ignored. This encoding is *optimistic*, i.e. it assumes that all results will be achieved. As an example, consider the translation of the SESMA markups from Fig. 2:

```
(:action op_buyItem
 :parameters (?item ?cc ?ccexp ?result)
 :precondition (and
   (have-creditcard_ ?cc ?ccexp)
   (in-catalog_ shop2 ?item))
 :effect (possess_ client ?item))
```

```
(:action op_getItemList
 :parameters (?title ?desc)
 :effect (forall(?item)
   (and (in-catalog_ shop2 ?item)
     (has-title_ ?item ?title)
     (has-description_ ?item ?desc))))
```

(ii) The WSC planning goal is translated into the precondition of a reserved operator whose effect is the planning goal; this allows us to use variables in goal

```
/* returns a boolean signaling success or failure. */
01 function SOLVE-PROBLEM(G, I, threshold, anno) {
02     solved ⇐ ⊥
03     attempts ⇐ 0
04     D ⇐ BUILD-DOMAIN(anno)
05     AL ⇐ EXTRACT-FROM(G)
06
07     label replan:
08     while( ¬solved ∧ attempts < threshold ) {
09         Plan π ⇐ MAKE-PLAN(D, I, G, AL) /* invoke VHPOP */
10         if(π = null) return ⊥
11         for each step s ∈ S(π) ordered by O(π) {
12             if(¬CAUSAL-LINK-VIOLATIONS(s, π, AL)) {
13                 EXECUTE(opr(s)) /* invoke service operation */
14             } else {
15                 attempts++
16                 continue replan
17             }
18         }
19     }
20     return (attempts < threshold)
21 }
```

Fig. 4. Main cycle of the replanning algorithm

descriptions. (iii) The fact base s_A of the planning agent is transformed into a set of ground atoms, the description of the initial state.

The next preparation step is to initialize the set $\mathcal{AL}$ with causal link patterns in order to avoid violations of the *find-out* constraints of the goal. This is done by calling the operation EXTRACT-FROM in line 5: For each literal l_g in the goal description $\mathcal{G}$ (which is represented in disjunctive clausal normal form) that has a *find-out*-annotation, all operations in the domain whose effects contain a literal l_{op} that unifies with l_g, a causal link pattern $\langle op, l_g, * \rangle$ is created and added to $\mathcal{AL}$. This way, the planner will avoid using steps that violate the *find-out*-constraints of the goal.

As described in Fig. 4, the algorithm then enters the main loop (line 8) and first calls the VHPOP routine MAKE-PLAN (line 9). If successful, VHPOP delivers a plan π which (i) is complete, (ii) may be partially ordered, and (iii) does not contain any causal links that match a causal link pattern in $\mathcal{AL}$. If the POP routine fails to come up with such a plan, then the process is terminated and reports failure (line 10).

The complete plan π devised by VHPOP has the *potential* to solve the given problem, but due to the nondeterministic nature of the domain, success is not *guaranteed*. Therefore, the agent is forced to expose the plan to the environment to learn whether or not a potential solution indeed achieves the desired goal. Therefore, the agent chooses a linearization L of $\mathcal{S}(\pi)$ which it aims to execute in a controlled manner (lines 11-18): For each plan step s, the func-

```
/* returns a boolean signaling causal link violations */
01 function CAUSAL-LINK-VIOLATIONS(Step s, Plan π, Set AL) {
02     boolean foundViolation ⇐ ⊥
03     for each causalLink cl = ⟨s_i, p, s_j⟩ ∈ L(π) {
04         if (cl is relevant) ∧ ¬(s_i achieves p) {
05             AL ⇐ AL ∪ cl
06             foundViolation ⇐ ⊤
07         }
08     }
09     return foundViolation
10 }
```

Fig. 5. Causal Link Violation Check

tion CAUSAL-LINK-VIOLATIONS(s, π, $\mathcal{AL}$) is called to test whether those preconditions of the remaining steps that should be already achieved are indeed fulfilled (cf. Fig. 5). It does so by testing whether the *relevant* causal links of the plan are fulfilled. A causal link $s_i \xrightarrow{p} s_j$ is *relevant* upon execution of step s if s_i was already performed and s_j is not performed yet, i.e. when $s_i \prec s \preccurlyeq s_j$ in L.

If all relevant causal links for s (and the steps beyond s) are fulfilled, then the agent will *execute* the operation represented by step s (cf. line 13 in Fig. 4). To execute an operation, the planner first creates the proper input message, where all variables are replaced by the intended values. The message is then sent to the Web service and its response is retrieved and parsed. For each result r in *Res(op)*, the success condition is tested, if one exists. If no success condition violation is detected, the result (i.e. effect or knowledge effect) gets materialized (cf. Sect. 3.5) and the agent's fact-base is updated. Note that the conversation data sets are used to substitute variables of effect formulas by constants.

After the execution of the current step of the plan is finished, the next step is chosen (cf. line 11 in Fig. 4), and the check for causal link violations is carried out again. If at some point in the plan execution a causal link violation is detected, then the plan execution is aborted (lines 15, 16). The causal links whose violations were detected by CAUSAL-LINK-VIOLATIONS remain in $\mathcal{AL}$ and will be used in the next planning-cycle.

As long as the threshold value is not reached (line 8), the replanning algorithm continues by generating another plan. Since the set $\mathcal{AL}$ contains information about causal link relationships that have turned out not to yield the desired results, the POP algorithm will avoid to rely on causal links that match any link in $\mathcal{AL}$. It will try to determine an alternative plan that avoids the mistakes of the previous attempt(s). If such an alternative plan is found, then the execution and monitoring of causal link violations starts again (lines 11-18).

Once the method SOLVE-PROBLEM manages to execute a plan *without* running into causal link violations, then the problem is *solved*.

6 Examples and Empirical Tests

For a brief empirical evaluation of the concepts presented in this article, consider a simple "web-shopping" domain, which consists of two online retailers A and B who offer several operations to browse their catalogs and to purchase items. The services and their WSDL & SESMA descriptions are available online at[1].

Shop A offers the following operations: *buyItem* purchases an item; the international EAN number of the item and user's credit card information must be given. Operation *getPrice* delivers the price of an item, and *getItemList* returns the list of items that are in the shop's catalog and can be purchased; each catalog item is described by title, EAN number and description. The operations *buyItem* and *getItemList* are depicted in Fig. 2.

Shop B has a different interface: it offers the method *getList* to retrieve a list of available items, described by EAN-number, title, price and description. Further, it uses a shopping cart metaphor for purchases: every item to be purchased needs to be put into a shopping cart first, by invocation of a method *addToCart*, which takes the EAN number of the item as input parameter. To remove an item from the cart, the method *removeFromCart* can be used, and to purchase the items in the cart the method *checkout* can be used, which requires credit card information as an input. Only registered users that are logged in and possess a valid session-ID for Shop B are permitted to use the *addToCart*, *removeFromCart* and *checkout* operations; to register at Shop B, the operation *register* needs to be called with information about the user, and to log in and retrieve a session token, the operation *login* needs to be called with the user credentials received from *register*.

Let us consider a scenario in which the goal *<possess item="123456" />* should be fulfilled, whereby the predicate *possess* belongs to the vocabulary used in the markup of the two shopping services. We implemented a prototype of the architecture and algorithms presented in this paper and ran it against 4 different test cases, which are listed in Tab. 1: The cases are different instantiations of the scenario described above; they differ regarding the availability of the product with EAN-number *123456*. In case nr. 1, the item is available at both shops, in case nr. 2, it is only available at Shop A, in case nr. 3 it is only available at Shop B and in case nr. 4, none of the shops have the item in their article catalog.

As the column 4 of the table shows, the SOLVE-PROBLEM algorithm terminates in all four example cases; even in the unsolvable case it does not take more than three planning attempts for the agent to come to the correct conclusion. Column 5 shows the total time the agent had to spend to solve the problem and col. 6 shows the amount of time used solely for the planning (which excludes the time spent for interaction with the Web services). Column 7 shows the final results of the experiments, i.e. whether or not the agent was able to purchase the desired product.

[1] http://wsplan.sf.net

Table 1. Example cases computed on Intel Pentium IV, 1.6 GHz, running Linux 2.4

Case Nr.	Shop A	Shop B	Terminates	Time (s) Total	Time (s) Planning	Result
1	Item available	Item available	Yes, after 1	1.65	0.11	Item purchased
2	Item available	Item not avail.	Yes, after 1	1.65	0.11	Item purchased
3	Item not avail.	Item available	Yes, after 2	1.98	0.17	Item purchased
4	Item not avail.	Item not avail.	Yes, after 3	1.73	0.24	Item not purchased

As an example, we will briefly describe case 3 of our test series, because it illustrates how the replanning algorithm reacts to adverse conditions: At first, an initial plan is created: [*ShopA.getItemList*, *ShopA.buyItem*]. The agent executes the first operation and retrieves the list of items available at Shop A, which does not contain the wanted product. Before it executes the next operation, it checks the causal links and detects that *getItemList* did not manage to achieve the precondition for *buyItem*, which is that the item is in the catalog. Therefore, the planner aborts the plan execution, adds the newly detected causal link violation to $\mathcal{AL}$ and then requests a new plan from the adapted VHPOP planner. The entry in $\mathcal{AL}$ makes it impossible for the planner to find a new plan based on Shop A. It is therefore forced to backtrack and to chose a different planning branch, i.e. to consider shop B. Therefore, in the second attempt, a plan [*ShopB.register*, *ShopB.login*, *ShopB.addToCart*, *ShopB.checkout*] is devised. In case 3, this attempt indeed achieves the goal because Shop B has the product in stock. We can see from the example, that with each planning- and execution-cycle the planner gains feedback about the plan structures that lead to dead ends, which gradually improves its ability to find potent plans that achieve the given goal.

7 Related Work

In recent time, several approaches to applying AI techniques to the WSC problem have been published. A relevant aspect that helps differentiating the various approaches is the way they represent domain knowledge. On the one hand, it is accepted that the proper encoding of domain knowledge is a key requirement for efficient planning. On the other hand, manually encoded domain knowledge may hamper practical adoption, because of the efforts and skills required.

Existing work like [9] and [10] use manually encoded domain knowledge, in the form of HTN method descriptions and GOLOG programs, respectively. Our solution does not use manually encoded domain knowledge, instead it uses knowledge generated by dynamic *interaction* with the domain. In this sense, we follow the tradition of *replanning* agents, where feedback data is gained to better inform the heuristics of the agent (cf. e.g. [11]). It must be admitted, however, that the domain knowledge gained from feedback of plan executions does not keep up to the richness of manually encoded knowledge. Probably a combination of both approaches would suite the WSC problem best.

Another central issue is the representation of *goals*. In this context, our work draws from SADL [12] which introduced goal annotations similar to the ones we use. However, our algorithmic strategy to deal with those goals differs from the existing work: we encode *find-out* goals into a set $\mathcal{AL}$ of forbidden causal link patterns and resort to replanning techniques, while the previous SADL-based approaches like PUCCINI [12] chose to deal with the problem by extending POP's plan-, domain- and problem representation schemes, including the introduction of new link- and threat-types. Our current use cases and experiments justify our strategy, but a systematic comparison of the advantages and disadvantages of each approach has yet to be conducted.

Another novel approach to service composition was presented in [13], which applies the planning as *model checking* (MC) paradigm. In this approach, the goal specifies the conditions that must hold after plan execution which can include conditions about the plan itself. The ability to pose such conditions (e.g. safety and liveness properties) on the plan is an advantage of the planning as MC approach. While our set $\mathcal{AL}$ of *links to avoid* also offers a tool to specify safety conditions, our approach does not allow for liveness properties. However, this solution comes with a certain computational overhead [13] which may lead to long planning times; more work is required to find out which approach serves which problem domains best.

8 Summary

In this paper we described how a modified version of a state-of-the-art partial order planner, embedded in an execution monitoring engine, can be used to automatically solve problems in Web service domains. We have illustrated how the agent can use feedback gained from failed plan executions to avoid doomed plans in the next planning attempts.

We found that the POP model of plans, especially the explicit representation of a plan's causal links provides several benefits: Firstly, it allows for execution monitoring that can detect plan failures even before they become immanent. Secondly, it allows for *precise* learning. For instance, in the example discussed in Sect. 6, the planner does not blindly avoid *all* links between *ShopA.buy* and *ShopA.getItemList*; it only avoids links between the two operations where the literal to be achieved unifies with the literal of the failed link. In practical terms, this means that some action is avoided to buy a product A because of a former failure, but an attempt to purchase a product B can still be made.

Another contribution of this work is the support for nondeterministic actions using *success-conditions*. These are extra-logical conditions that can be attached to effect descriptions and are evaluated *after* an operation has been carried out. This way, agents can execute nondeterministic actions and easily determine the state they are in afterwards.

In future work we aim to clarify the issues raised in Sect. 7. Further, we plan to study extensions of the current *avoid-links* concept and to investigate *compensation actions* that may be required after failed attempts.

Acknowledgements

The author would like to thank Biplav Srivastava and Jana Koehler for their valuable feedback on earlier versions of the paper, Håkan Younes for the permission to extend VHPOP and the anonymous reviewers for their helpful comments.

References

1. W3C: Web Services Description Language (WSDL) Version 1.2 (2002)
2. Srivastava, B., Koehler, J.: Web Service Composition - Current Solutions and Open Problems. In: Proceedings of the ICAPS'03 Workshop on Planning for Web Services (2003)
3. Carman, M., Serafini, L., Traverso, P.: Web Service Composition as Planning. In: Proceedings of the ICAPS'03 Workshop on Planning for Web Services (2003)
4. OWL-S Coalition: OWL Web Services 1.1, http://www.daml.org/services (2004)
5. WSML Working Group: Web Service Modeling Language (WSML), http://www.wsmo.org/wsml (2004)
6. Peer, J.: Semantic Service Markup with SESMA. Language Specification, version 0.8, http://elektra.mcm.unisg.ch/pbwsc/docs/sesma_0.8.pdf (2004)
7. Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., Williamson, M.: An Approach to Planning with Incomplete Information. Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (1992)
8. Younes, H.L.S., Simmons, R.G.: VHPOP: Versatile Heuristic Partial Order planner. Journal of Artificial Intelligence Research (2003)
9. Hendler, J., Wu, D., Sirin, E., Nau, D., Parsia, B.: Automatic Web Services Composition Using SHOP2. In: Proceedings of The Second International Semantic Web Conference(ISWC) (2003)
10. McIlraith, S., Tran, C.S., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems (2001)
11. Haigh, K.Z.: Situation Dependent Learning for Interleaved Planning and Robot Execution (1998)
12. Golden, K.: Planning and Knowledge Representation for Softbots (1997)
13. Traverso, P., Pistore, M.: Automated Composition of Semantic Web Services into Executable Processes. In: Third International Semantic Web Conference ISWC'04. (2004)