Generating workflow models from OWL-S service descriptions with a partial-order plan construction

Bochao Wang
Australian National University
u4465961@anu.edu.au

Armin Haller, Florian Rosenberg
CSIRO ICT Centre
firstname.lastname@csiro.au

Abstract

In this work we construct partial order plans from a pool of atomic services described in OWL-S. We make extensions to Partial Order Planning to allow multiple conditional effects in action definitions. The purpose is to handle the uncertain behavior of Web services with incomplete initial information. We post-process the partial order plan to auto-generate a workflow model. We developed a method to identify a subset of workflow patterns from the solution plan to create a workflow diagram.

1 Introduction

The Service-oriented computing paradigm has enabled an increasing number of companies to deploy and remotely access functionality exposed as Web service. However, the true potential of a distributed infrastructure can only be reached when existing services can be composed into more complex services to meet the requirements of a user automatically. In this paper we introduce a semi-automatic service composition framework that produces a workflow plan from semantically annotated Web service. In particular, we propose extensions to Partial Order Planning to handle uncertain behavior of Web services with incomplete initial information. The planner builds a partial order plan that splits parallely, conditionally and also joins. Similar to [9], we use a forward planning technique from an initial state of the world. Then, we post-process the plan produced by the planner to auto-generate a workflow representation. We develop a method to identify a subset of workflow patterns from the solution plan to create a workflow diagram. The resulting graphical workflow diagram can be used by a designer to get an intuitive understanding of the high level business logics of the process or can be used to be directly mapped to an operational business modeling languages such as WS-BPEL.

2 Constructing a Partial Order Plan

In our approach we make adaptations to Partial Order Planning for service composition and refer to the developed planner as SC-APOP. Our planner is similar to basic POP such as [8], with a major difference - action. We enable multiple conditional effects. The output solution also includes a Context-Table, in addition to a partial order plan. The definitions of the planning domain are as follows:

Definition (Planning Domain): A Planning Domain is a tuple $D = \langle S, A \rangle$, where:

- S is a set of predicates;
- A is a set of actions.

Definition (Planning Problem): A Planning Problem α is a tuple $\alpha = \langle D, I, G \rangle$, where:

- D is a planning domain.
- I, G are initial and goal states respectively. They are both conjunction of predicates.

Definition (Predicate): A *Predicate*, S is a string name and a set of variables or constants, each of which is of a type.

Definition (Action): An action A is a tuple $\langle P, C, E \rangle$, where:

- P is the precondition of the action. P is a conjunction of predicates, $S_i \wedge ... \wedge S_j$. P can be null.
- C is a set of secondary preconditions $C_i, i = \{1, ..., l\}$. Each secondary precondition C_i is a conjunction of predicates, $S_i \wedge ... \wedge S_j$. C_i has to be incompatible with C_k where $i, k \in \{1, ..., l\}$ and $i \neq k$. C can be null.
- E is a set of effects $E_i, i = \{1, ..., l\}$. Each effect E_i is a conjunction of predicates, $S_i \wedge ... \wedge S_j$. E_i has to be incompatible with E_k where $i, k \in \{1, ..., l\}$ and $i \neq k$.

- C_i and E_i are a pair of secondary precondition and conditional effects.

An action represents an atomic service. As indicated in the above definition a service may have multiple conditional effects. We restrict a precondition P, secondary precondition C_i and effect E_i to conjunctions of predicates. Precondition P and/or secondary precondition $C_i, i = \{1, ..., l\}$ can be empty. We further define the notion of compatible and incompatible. Two predicates S_1, S_2 are *compatible* (*incompatible*) iff the they can (not) hold simultaneously. In addition, two sets of predicates X, Y are compatible iff each predicate in X is pairwise compatible with each predicate in Y.

Definition (Set of Steps): A step a is a ground action when it appears in a plan. We use T to denote a set of steps.

Definition (Causal Link): A *causal link* is a tuple $\langle f, S, h \rangle$, denoted by $f \xrightarrow{S} h$, where S is a predicate, f is a step name that has S in its effect E_i, h is a step name that has S in its precondition P or secondary precondition C_j. A causal link $f \xrightarrow{S} h$ indicates that executing step f establishes a precondition of h. We say f is an *establisher* of h [11].

Definition (Threat): A step name q is called a *threat* to a causal link $f \xrightarrow{S} h$ if q is a step other than f and h, that has $\neg S$ in its effect.

Definition (Ordering Constraint): An *ordering constraint* $f \prec h$ indicates that step f must precede h in the plan. Each causal link is associated with an ordering constraint.

Definition (Partial Order Plan): A Partial Order Plan π is a tuple $\pi = \langle L, O, T \rangle$, where:

- L is a set of causal links;
- O is a set of ordering constraints;
- T is the set of steps.

Definition (State Context): *State Context* is a set of ground predicates. A state context of an action represents the set of predicates that hold while the action begins to execute.

Definition (Context-Table): A *context-table* CT keeps the state context for each step a. We denote the context of a by $CT(a)$.

Basic POP works by iteratively adding steps, causal links, ordering constraints and binding constraints until the plan is *complete* [8]. It does not restrict the order of planning as long as a complete plan is found. SC-APOP adopts forward, ordered planning in which steps are included into the plan in an order of some valid linearization of the plan to be completed later. The implementation:

1. keeps a closed list for action steps that has been already "executed". The closed list is the set of steps T of the plan π.
2. keeps an open list of steps. An element is inserted if all its preconditions can be established by steps that have been already in the plan. If a candidate action to be expanded has any precondition where we can not find an establisher in current π, the candidate shall be reached and added at a later stage when its other establishers are in the plan.

Figure 1 shows our algorithm in detail. We assume a set of ground predicates and actions are created and filtered beforehand. Line 8 is the goal test. Line 12-18 checks whether the establishers of a candidate step are in the current plan. Line 20-24 insert causal link and ordering constraints for the new step. Then threats are resolved by promotion or demotion [10] in line 26. We then update the context-table in the sub-routine `UPDATE-CONTEXT-TABLE()`, where the context of the new step is the union of that of its establishers, as shown in line 41,43. In Line 36, if we can no longer include more steps into the plan while the dummy END action is still not found, the planning problem has no solution. After a new step is included into the plan with all associated causal links, we resolve all threats in the current plan immediately.

3 Constructing Workflow Diagrams

In our approach to map a partial order plan produced by SC-APOP to a set of workflow patterns [15] we address the five basic control flow patterns - sequential, parallel-split, synchronization, exclusive-choice, simple-merge and multiple-choice, synchronization-merge of the advanced branching and synchronization patterns.

Convert Partial Order Plan to DAG: The partial order plan generated from SC-APOP can be seen as a Directed Acyclic Graph (DAG). Any valid partial order plan can be converted straightforwardly into a DAG $\langle V, E \rangle$ where: V is a set of vertices and E is a set of edges. Each vertex v_i corresponds to an action step t_i and each edge e_j corresponds to an ordering constraint o_j of the conditional partial order plan. There is a source-vertex o that corresponds to the START step and a sink t node that corresponds to the END step.

Transitive Reduction: The method of mapping a partial order plan to the set of identified workflow patterns treats each ordering constraint as the control flow of the workflow diagram/patterns. To reduce the complexity of the workflow diagram to be generated while preserving the necessary execution flow, we first perform a Transitive Reduction [1] on the direct graph converted from the plan.

Post-Processing Output: On completion of the post-processing procedure, another graph (a workflow diagram) is generated with the following vertex-types.

Definition (Vertex-Type): *Vertex-Type* is an attribute associated with each vertex. Its ranges are parallel-split,

```
0: SC-APOP( α )
1:   Let START be a_0, and manually specified actions
       with no preconditions be a_1, ..., a_m
2:   OPEN.enqueue( a_0, a_1, ..., a_m )
3:   π.T = π.T ∪ ( a_0, a_1, ..., a_m )
4:   Create empty context-table CT
5:   CT(a_0) = I
6:   WHILE OPEN ≠ ∅
7:     a_x = OPEN.dequeue()
8:     IF a_x == END, report success and return π
9:     FOR each predicate S_u in each conditional effect
         E_i of a_x
10:      FOR each action a_y ∈ A
11:        FOR each predicate S_v in each secondary
             precondition and precondition E_j of a_y
12:          IF S_u, S_v are same predicate and consistent
13:            allPreEstablishable = true
14:            FOR each predicate S_w in the precondition
                 P and each secondary precondition
                 C_k of a_y
15:              IF there does not exist a causal link
                   of the form a_z -S_w-> a_y in π.L
16:                allPreEstablishable = false
17:              ENDIF
18:            ENDFOR
19:            IF allPreEstablishable
20:              FOR each predicate
                   S_w in the precondition
                   P and each secondary precondition
                   C_k of a_y
21:                FOR each causal link of the form
                     a_z -S_w-> a_y in π.L
22:                  π.L = π.L ∪ (a_z -S_y-> a_y)
23:                  π.O = π.O ∪ (a_z ≺ a_y)
24:                ENDFOR
25:              ENDFOR
26:              Resolve threats
27:              Call UPDATE-CONTEXT-TABLE()
28:              OPEN.enqueue(a_y)
29:              π.T = π.T ∪ a_y
30:            ENDIF
31:          ENDIF
32:        ENDFOR
33:      ENDFOR
34:    ENDFOR
35:  ENDWHILE
36:  RETURN Failure

37: UPDATE-CONTEXT-TABLE()
38:  FOR each ordering constraint of the form a_r ≺ a_y
       where a_r ∈ π.T
39:    IF CT(a_y) and CT(a_r) are incompatible
40:      Let S_s, S_t be any incompatible predicates in
           CT(a_y) ∪ CT(a_y)
41:      CT(a_y) = CT(a_y) ∪ CT(a_r) − S_s − S_t
42:    ELSE
43:      CT(a_y) = CT(a_y) ∪ CT(a_r)
44:    ENDIF
45:    IF a_r ≺ a_y is also a causal link a_r -S_o-> a_y
46:      CT(a_y) = CT(a_y) ∪ precondition P of a_y
47:      CT(a_y) = CT(a_y) ∪ S_o
48:    ENDIF
49:  ENDFOR
```

Figure 1: SC-APOP algorithm

synchronization, exclusive-choice, simple-merge, multiple-choice, synchronization-serge and action-step.

Definition (Post-Processing Output): Post-Processing Output is another DAG $\langle V', E' \rangle$ where:

V' is a set of vertices and E' is a set of edges. Each $v \in V'$, v has a vertex-type. $\langle V', E' \rangle$ is any valid workflow diagram.

3.1 Post-Processing Procedure

1. Convert Partial Order Plan to DAG and remove the sub-graph irrelevant to the goal by backward search for relevant edges from the sink to the root.
2. Perform transitive reduction to the DAG.
3. For each vertex v_k:

(a) Recognize parallel-split. Suppose the recognized parallel-split is $f_k \prec h_i$, $i = \{1, ..., j\}$. Remove edges (f_k, h_i), $i = \{1, ..., j\}$. Add vertex v of type parallel-split. Add edges (f_k, v) and $(v, h_i), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $f_k \prec h_i$, $i = \{1, ..., j\}$.

(b) Recognize synchronization. Suppose the recognized synchronization is $h_i \prec f_k$, $i = \{1, ..., j\}$. Remove edges (h_i, f_k), $i = \{1, ..., j\}$. Add vertex v of type synchronization. Add edges (v, f_k) and $(h_i, v), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $h_i \prec f_k$, $i = \{1, ..., j\}$.

(c) Recognize exclusive-choice. Suppose the recognized exclusive-choice is $f_k \prec h_i$, $i = \{1, ..., j\}$. Remove edges (f_k, h_i), $i = \{1, ..., j\}$. Add vertex v of type exclusive-choice. Add edges (f_k, v) and $(v, h_i), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $f_k \prec h_i$, $i = \{1, ..., j\}$.

(d) Recognize simple-merge. Suppose the recognized simple-merge is $h_i \prec f_k$, $i = \{1, ..., j\}$. Remove edges (h_i, f_k), $i = \{1, ..., j\}$. Add vertex v of type simple-merge. Add edges (v, f_k) and $(h_i, v), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $h_i \prec f_k$, $i = \{1, ..., j\}$.

(e) Recognize multiple-choice. Suppose the recognized multiple-choice is $f_k \prec h_i$, $i = \{1, ..., j\}$. Remove edges (f_k, h_i), $i = \{1, ..., j\}$. Add vertex v of type multiple-choice. Add edges (f_k, v) and $(v, h_i), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $f_k \prec h_i$, $i = \{1, ..., j\}$.

(f) Recognize synchronization-merge. Suppose the recognized synchronization-merge is $h_i \prec f_k$, $i = \{1, ..., j\}$. Remove edges (h_i, f_k), $i = \{1, ..., j\}$. Add vertex v of type synchronization-merge. Add edges (v, f_k) and $(h_i, v), i = \{1, ..., j\}$. Replace f_k by v in ordering constraints $h_i \prec f_k$, $i = \{1, ..., j\}$.

4 Related work and Conclusion

In this work we made extensions to Partial Order Planning (POP) to allow multiple conditional effects in action definitions. The algorithm is able to keep the business or system state for each service in context tables, which are used at the next stage when recognizing workflow patterns. A very similar planning technique, called Hierarchical Task Network (HTN) planning, was shown to have various advantages over the other AI planning techniques because of its scalability and concurrency [5]. Using HTN planner SHOP2 for web service composition was reported in [13]. The basic HTN planning algorithm are similar to POP with one important exception that HTN operations can be both primitive actions, which corresponding to the usual actions in the POP, and non-primitive actions [6]. Although HTN planner enables a hierarchical description of composite services, the assumption made by this mechanism is that the service has already been composed, which makes no contribution to the notion of automatic composition. State transition systems are another AI planning technique that has been used for Web service composition [3; 14; 12].

Alternative approaches to partial order planning are for example Planning Graph Analysis [4] and Planning as Sat-

isfiability [7]. The use of such planning approaches for Web service composition is proposed in [16] where the authors present an algorithm based on planning graph model and a set of strategies for pruning redundant Web Services. Web service composition involves various challenges in a number of aspects that are not addressed in our work. For example, [2] address the issue of inexact terms in semi-automated web service composition.

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] R. Akkiraju, B. Srivastava, A.-A. Ivan, R. Goodwin, and T. Syeda-Mahmood. SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition. In Proc. of *ICWS 2006*, pages 37–44, 2006.

[3] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of Web services via planning in asynchronous domains. *Artif. Intell.*, 174:316–361, March 2010.

[4] A. L. Blum and M. L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[5] K. S. M. Chan, J. Bishop, and L. Baresi. Survey and Comparison of Planning Techniques for Web Services Composition. Technical report, Univ. of Pretoria, 2007.

[6] S. Kambhampati. A Comparative analysis of Partial Order Planning and Task Reduction Planning. *SIGART Bull.*, 6:16–25, January 1995.

[7] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proc. of AAAI-96*, pages 1194–1201, 1996.

[8] D. Mcallester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proc. of AAAI-91*, pages 634–639, 1991.

[9] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[10] J. S. Penberthy and D. S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. pages 103–114. Morgan Kaufmann, 1992.

[11] M. A. Peot and D. E. Smith. Conditional nonlinear planning. In *Proc. of ICAP 92*, pages 189–197, 1992.

[12] M. Pistore, P. Roberti, and P. Traverso. Process-level composition of executable Web services: on-the-fly versus once-for-all composition. In *Proc. of ESWC05*, pages 62–77, 2005.

[13] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for Web Service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.

[14] P. Traverso and M. Pistore. Automated composition of semantic Web services into executable processes. In *Proc. of ISWC 2004*, pages 380–394, 2004.

[15] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. 14(1):5–51, 2003.

[16] X. Zheng and Y. Yan. An efficient syntactic Web service composition algorithm based on the planning graph model. In *Proc. of ICWS*, pages 691–699, 2008.