Learning Non-deterministic Action Models for Web Services from BPEL4WS Programs

Dongning Rao[1], Zhihua Jiang[2,1], Yunfei Jiang[1]

[1]Software Research Institute, School of Information Science and Technology, Sun Yat-Sen University, Guangzhou 510275, P.R. China
[2]Department of Computer Science, Jinan University, Guangzhou 510632, P.R. China
E-MAIL: rdn2006@163.com, jnujzh@163.com, issjyf@mail.sysu.edu.cn

Abstract

Planning is a promising technique for Web Service Composition (WSC). As industrial engineers and academic researchers use different languages, we try to bridge the gap between them by extracting necessary models for planning from existing industrial solutions. In the mean time, Web Services (WS) may have multiple outcomes and we need to learn those possible outcomes. So in this paper, we address the problem of learning non-deterministic action models for Web Services from existing WSC solutions which are the Business Process Language for Web Services (BPEL4WS) programs. To do so, we first provide methods to translate BPEL4WS programs into a planning domain. This can be very helpful for verifying the WSC solution in BPEL4WS program too. And then because of the non-deterministic nature of Web Service we further extend the scope of learning action models into non-deterministic planning (NDP). Finally we test our approach on some samples provided by a popular BPEL4WS tool. This work has interesting consequences both from a practical and a theoretical point of view, and it can shorten the distance between AI planners and the real world.

Introduction

Could we always get our works done by a single Web Service? No. To remedy this, people brought WSC out. Currently, WSC is addressed by two orthogonal efforts: the business world has developed BPEL4WS (IBM *et al.*, 2002) and WS interfaces are like remote procedure call and the interaction protocols are manually written; academic society draws their attention on WSC as AI planning e.g. (McIlraith & Fadel, 2002), (McDermott, 2002), (Wu *et al.*, 2003), (Martínez & Lespérance, 2004), (Peer, 2004), (Vukovic & Robinson, 2004), current advance and some open problems are discussed in (Srivastava *et al.*, 2003).

Unfortunately, although planning is one of the most promising techniques for WSC, the problem corresponding to WSC as planning is far from trivial. Obstacles such as knowledge-engineering bottlenecks stand in front of us from the beginning: manual construction of action models for domain descriptions is tedious and painstaking, even for experts in planning community. As industry engineers are not familiar with Planning Domain Definition Language (PDDL, which is a basic format of the inputs of most planning tools), most existing WSC solutions are written in BPEL4WS. But when we use planning tools we expect action models to be described in PDDL. It is also difficult for planning researchers to describe Web Service with PDDL as they are not familiar with real world applications. So learning action models is useful and designing them by hand is not desirable.

Building on these insights, in this paper we address the problem of learning non-deterministic action models for Web Services from existing WSC solutions which are BPEL4WS programs. Instead of building WSC solutions from scratch we try to learn action models from existing solutions. As far as we know, it is the first time that BPEL4WS Executable processes[1] are translated into planning domains. This is also the first attempt to describe states translated from BPEL4WS programs in proposition set level. In the mean time, we first extend the scope of learning action model into NDP. Our work will eventually (not at current stage) relieve people from writing planning description for existing services, and make planning tools more applicable for real-world WSC problems.

The remainder of this paper is organized as follows. For a better understanding for the background, we focus on explaining web service composition as planning, BPEL4WS and previous works about planning with BPEL4WS in the next section. A briefly introduction of learning action models will also be presented there. The architecture of our whole process will be showed in the Architecture section along with a pedagogical example. Detailed descriptions of the components will follow. Finally, in the last section, we summarize the paper with experiment results and differences with related works, future research directions will be identified there too.

[1] BPEL4WS Process is a container where you can declare the activities to be executed and so forth. These processes are written as programs before being executed, and in this paper we will use BPEL4WS process and BPEL4WS program as interchangeable terms. BPEL4WS Process can be divided into two categories: Executable Process or Abstract Processes,, Abstract Processes can define the interfaces of WS, but only Executable BPEL4WS processes can address WSC.

Background and Related Work

Web Service Composition as Planning

The task of WSC is to automatically sequence together Web Services into a composition that achieves some user-defined objectives. As the industrial way, BPEL4WS, is primarily syntactical, planning as a promising technique has gained more and more interests from academic world.

Early works in this research line looked WSC as complex planning action composition such as in (McIlraith & Fadel, 2002), (McDermott, 2002), and (Peer, 2004). Later, researchers start to consider more and more real world conditions and combine tools other than planners such as (Martínez & Lespérance, 2004) uses knowledge database and (Vukovic & Robinson, 2004) tries to let WSC be context awareness. Many recent works like (Wu *et al.*, 2003) and (Kuter *et al.*, 2004) assume that there exist semantic descriptions for services, and provide translation methods from semantic descriptions to planning action models. Unfortunately under most circumstances such descriptions are unavailable in real world applications.

(Vukovic & Robinson, 2004) says: "By describing a Web service as a process in terms of inputs, outputs, preconditions and effect, using the metaphor of an action, composition can be viewed as a planning problem.", and the same idea is the basis of most papers in this research line, e.g. (McIlraith & Fadel, 2002), (McDermott, 2002), (Wu *et al.*, 2003) and (Martínez & Lespérance, 2004). (Wu *et al.*, 2003) further reveals that web services have unpredicted nature inherited from the internet, so it must be modeled with nondeterministic behaviors, and planning algorithms must work with uncertain effects. In this paper we will follow these works and model Web Services as actions with nondeterministic effects.

The research line that involves actions with nondeterministic effects is NDP, which has been devoted to increasing interests and several extensions of PDDL have been proposed. Such as NADL+ (Jensen & Veloso, 2000), NPDDL(Bertoli *et al.*, 2003), and PPDDL (Younes & Littman, 2004). But as none of them is the standard specification and in this paper we only care about proposition sets as preconditions and effects, we choose to state an action with mathematic formulas like tuples instead of in a planning language.

BPEL4WS

BPEL4WS was first conceived in July, 2002 with the release of the BPEL4WS 1.0 specification (IBM *et al.*, 2002), a joint effort by IBM, Microsoft, and BEA. BPEL4WS has been designed specifically for WSC, both for the publishing and the execution of compositions. So soon after BPEL4WS was proposed, it became the industrial standard.

As BPEL4WS became more and more popular, researchers pay more and more attentions on it. (Foster *et al.*, 2003) discusses a model-based approach to verify WSC, it tries to compile BPEL4WS programs into a Finite State Process notation (FSP) to allow an equivalence trace verification process to be performed. (Fu *et al.*, 2004) presents a technique for analyzing interactions of composite web services which are specified in BPEL4WS format, by translating a BPEL4WS Abstract Process into a guarded automata and then verifying it using a finite state model checker. The planning community gives its responses too. (Pistore *et al.*, 2004) and (Pistore *et al.*, 2005) consider services that are specified and implemented in BPEL4WS Abstract Process and generate plans for WSC. (Traverso & Pistore, 2004) generates plans for WSC[2] and than translates it into BPEL4WS programs.

These papers show that there is equivalence between plans and BPEL4WS programs, so we stand on the shoulders of the giants and view BPEL4WS Executable Process as a plan and then try to extract action models, which are corresponding with planning descriptions of Web services. By reusing these methods in (Foster *et al.*, 2003), (Fu *et al.*, 2004), and (Pistore *et al.*, 2005) we can translate BPEL4WS programs into a plan.

Learning Action Models

In the last two decades, for AI planning systems require the definition of action models, and it happen to be one of the most difficult tasks to build action models from scratch, there came learning action models (LAM). Hence various approaches have been explored to learn action models from examples. (Wang, 1995) first learns planning operators by observing expert solution traces; (Balac *et al.*, 2000) uses regression trees to learn action models; (Blythe *et al.*, 2001) acquires action models based on a procedure in which a computer system interacts with a human expert; (Pasula *et al.*, 2004) presents an algorithm for learning probabilistic STRIPS-like planning operators. These works are successful for fully observable domains and deterministic actions. Recently, there are new movements in this research line, (Shahaf *et al.*, 2006) is heading for partial observed environment and (Yang *et al.*, 2007) tackles incomplete information, but their work is still limited in the scope of deterministic action. Statistical methods and logical inferences are basic tools in these works.

Previous works like (Shahaf *et al.*, 2006) and (Yang *et al.*, 2007) motivated us to further extend the LAM scope. In the mean time although NDP is developed dramatically these years, there are still no previous works on learning non-deterministic action models until now. So in this paper we try to address the problem of learning non-deterministic action models under fully observations.

Architecture

Our goal is to automatically generate action models from existing service compositions. Here each action is corresponding to a specific Web Services W_i.

[2](Traverso & Pistore, 2004) generates plans from semantic descriptions of Web Service, and this description has nothing to do with BPEL4WS.

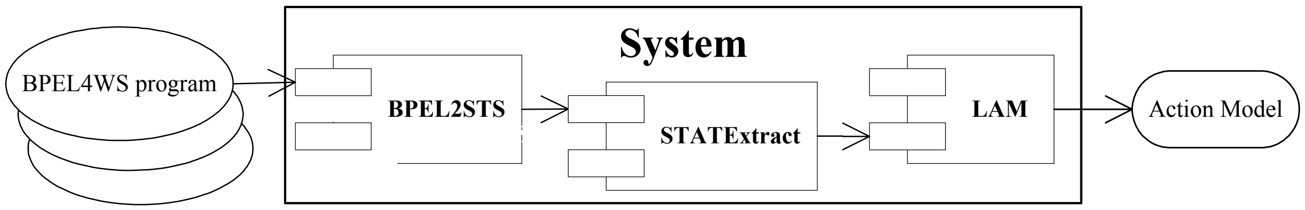


Figure 1: System Overview

More specifically (see Figure 1), we assume that service compositions are described as BPEL4WS Executable processes. Given n BPEL4WS Executable processes $B_1, \ldots, B_n$, the BPEL2STS module automatically translates each of them into a state transition system (STS from now on), $\Sigma B_1, \ldots, \Sigma B_n$. Intuitively, each ΣB_i is a compact representation of all possible behaviors, evolutions of the service composition B_i. Each ΣB_i is described in terms of states, actions, and transitions.

We then extract STate-Action-sTate (STAT from now on[3]) tuples from every ΣB_i. This is done by STATExtract module. Finally all STAT about the same action[4], are inputted to LAM (Learn Action Model) module. The LAM module will output the action model. The following pedagogical example will be used throughout the paper.

Example 1: There are two simple[5] solutions. They both call the same Web Service: BookRoom. This BookRoom Web Service is provided by a hotel, and will return success if there are available rooms and no internal errors in the hotel's system (namely BookSuccess = true), else it will return failure (namely BookFail = true). Suppose solutions are WSC_A and WSC_B. WSC_A is TryToBookRoom solution (See Figure 2), it calls BookRoom Web Service and set BookedFlag if success, and do nothing else. WSC_B is MakeSureToBookRoom solution (See Figure 3), and it calls BookRoom Web Service until success.

These solutions can be manually written in BPEL4WS programs[6] (See Figure 4 and Figure 5).

The architecture of the whole process has been shown in this section, along with a pedagogical example. Detailed descriptions of the components follow in the next three sections. We will state the BPEL2STS module in the next section, and the method which is used to extract STAT will be presented after that. The way to learn non-deterministic action models will be addressed at last.

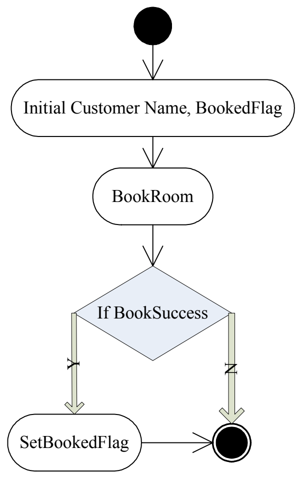


Figure 2:WSC_A Flow Chart

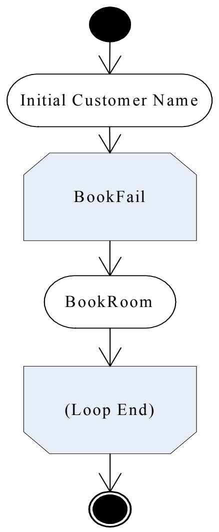
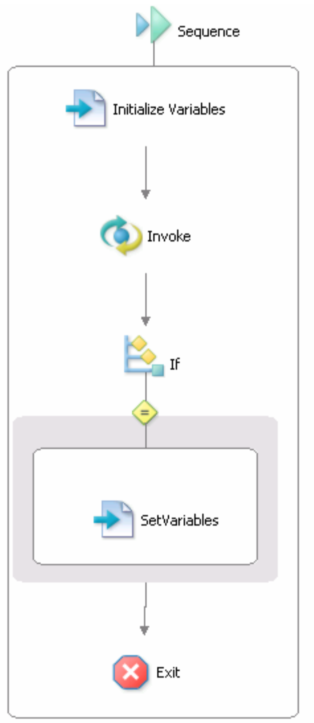


Figure3:WSC_B Flow Chart



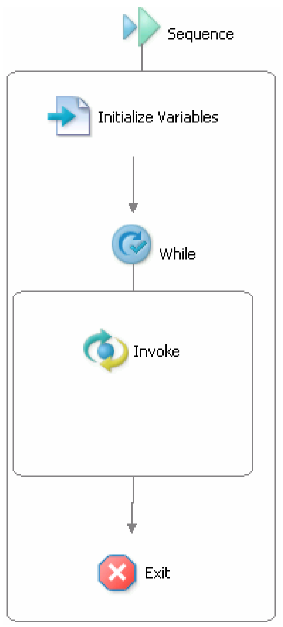


Figure 4:BPEL4WS WSC_A Figure 5:BPEL4WS WSC_B

From BPEL4WS to STS

In this section we will present a method to translate BPEL4WS programs into STS. We will first restate the major elements in BPEL4WS programs and review the

[3]SAS sounds better, but it is has been used by (Sandewall & Ronnquist, 1986) as an acronym for Simplified Action Structures.

[4]Here we assume the same action will have the same name in different programs, and vice versa.

[5]WSC_A is a sample of condition structure and WSC_B is a sample of loop structure.

[6]With ActiveBPEL, which is the most popular open Source BPEL4WS engine, see http://www.activebpel.org.

previous related works because our work is built heavily on them. And then we will address the translation of condition structure and loop structure, which is the first attempt as far as we know.

The major activities of providing and consuming WS in a BPEL4WS[7] process include: invoke, receive, and reply. Other activities in BEPL4WS programs are: assign, throw, wait, terminate, empty, catch, compensate, and exit. These activities are structured by: sequence, flow, switch, while, pick, if, repeatUtil, forEach, link. Additionally, BPEL4WS processes use < partnerLink > to define WS to be invoked and declare variables with < variable >. < partnerLink > are defined in the Web Service Description Language (WSDL) (W3C, 2002). (Foster *et al.*, 2003) first translates a subset of BPEL4WS elements into FSP, including sequence, switch, while, pick and flow; (Fu *et al.*, 2004) projects BPEL4WS elements into a guarded automata, including: assign, receive, invoke, sequence and flow; and (Pistore *et al.*, 2005) has handled BPEL4WS basic and structured activities, like invoke, receive, sequence, switch, while, flow (without links) and pick.

Among those unsettled activities, some will not affect the state in a system, like reply, wait, terminate, empty, and exit, and we will not discuss them here; some are about exception handling, like throw, catch, and compensate, which we believe can be translated into condition structures[8] . So in this paper we try to address condition structure (switch or if) and loop structure (while or repeatUntil)[9] , and we will just use if and while as example.

Another issue we have to address is that none of these previous works has described states in proposition set level. Previous works only need to tell states apart, but we have to know which proposition is true in a state. So we first define proposition space as follows:

Definition 1 (proposition space) A proposition space *Prop* is the set of all possible propositions in a system.

Then we can further define STS as (Pistore *et al.*, 2005):

Definition 2 (State transition system (STS))

A *state transition system* Σ is a tuple<S, S_0, A, R, L> where:

- S is the finite set of states;
- $S_0 \subseteq S$ is the set of initial states;
- A is the finite set of actions;
- $R \subseteq S \times A \times S$ is the transition relation;
- L: $S \rightarrow 2^{Prop}$ is the labeling function.

To build STS, we first[10] have to find out the *Prop* (by PropCollection module), and then the BPEL4WS program can be handled by methods provided in previous works, especially (Pistore *et al.*, 2005), to find out all states and transitions. After this, we have to further split states into if structure and while structure in SplitState module. Finally we build up the labeling function *L* by BuildL module (See Figure 6).

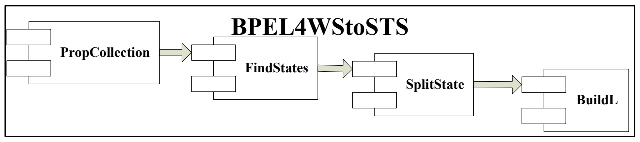
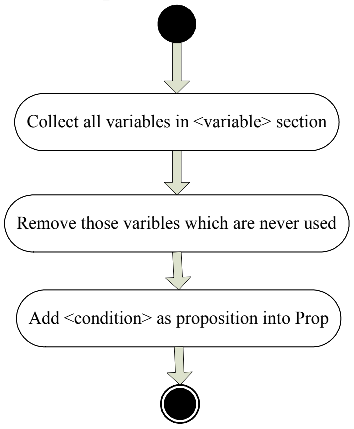


Figure 6: BPEL4WStoSTS

PropCollection works as Figure 7. Removing unused variables is for less redundancy of the STS, and this will lead to less redundancy of the learned action model. Most often, the conditions in if and while structure (in <condition> section) are only part of a variable, here we use this part of a variable as an independent proposition.


Collect all variables in <variable> section
Remove those varibles which are never used
Add <condition> as proposition into Prop

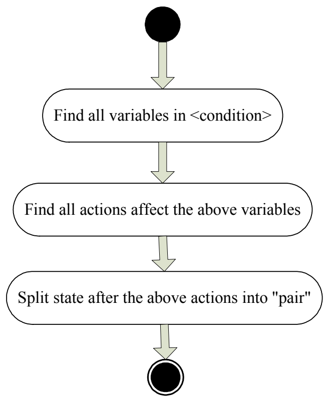

Find all variables in <condition>
Find all actions affect the above variables
Split state after the above actions into "pair"

Figure 7: PropCollection Figure 8: SplitState

FindStates module is designed following BPEL2STS module in (Pistore *et al.*, 2005), and we refer the readers to their work for more details. Simply put, here we obtain all states in the system. Example 2 includes an illustration of this process.

The SplitState module works based on the following assumption: there must be an action which affects the variable that is used in <condition>. Its flow is illustrated in Figure 8. Here the states come from the same action is a *pair* and the one we made up is the *pair state* of the original state.

Finally, in BuildL (Build Labeling function) module, we first let all propositions be false in the initial state and then change the propositions according to the transitions (result of FindStates). We assign different values to the proposition corresponding to the condition to the pair stat*e*.

Example 2: The results of PropCollection for WSC_A in Example 1 are:

$Prop_{WSCA}$= {CustomerName[11] , BookedFlag, BookSuccess}

Then FindStates will work like Figure 9, and SplitState will work like Figure 10.[12]

[7]According to the newest version BPEL4WS 2.0 (OASIS, 2007).

[8]This work remains undone, and it should be one of our future works.

[9]forEach and link structure are special, and we left them as one of our future works (link structure could be very useful for us).

[10]Here we suppose all actions have been listed, this can be done by look into the partnerLink part in BPEL4WS programs. In fact, more information can be found in the corresponding WSDL file. (Peer, 2004) and (Lerman *et al.*, 2006) have provide interesting results, and we are planning to combine their ideas into our work later.

[11]CustomerName is a Boolean flag, not a variable.

[12]The results of BuildL are proposition sets which represent states, we choose to omit details here for the concise of this paper.

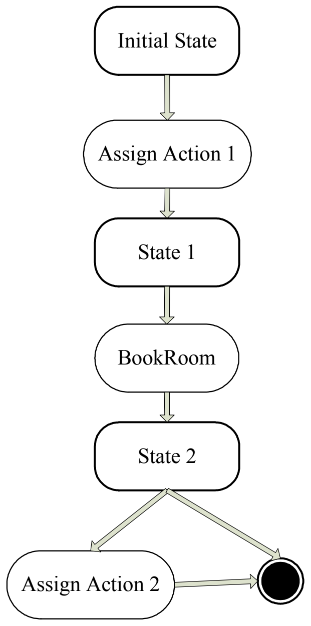


Figure9: FindStates Result

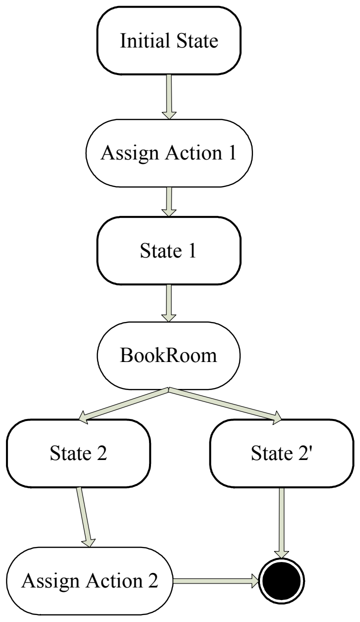


Figure 10: SplitState Result

Extracting State-Action-State Tuples

Previous works look a Web Service as a planning action, and they even provide the theorems about the equivalence of WSC/planning problem (Wu *et al.*, 2003), the application evolution/planning state evolution (Pistore *et al.*, 2005), and the WSC solution/plan (Wu *et al.*, 2003; Pistore *et al.*, 2005). In this section, we will further project BPEL4WS programs into plans in NDP as (Wu *et al.*, 2003) and (Pistore *et al.*, 2005). After this projection we view STAT as the possible outcomes for actions in a NDP system and extract STAT from STS.

Following (Cimatti *et al.*, 2003), we define planning domain, planning problem and plan as follows:

Definition 3 (planning domain) A *nondeterministic planning domain* is a 4-tuple D = < P; S; A; T >, where:
P is the set of propositions;
S is the set of states;
A is the set of actions;
T: S×A×S is the transition function. It associates to each current state s $\in$ S and to each action a $\in$ A the set T(s, a) $\subseteq$ S of next states.

Here we can see that STS in Definition 2 is a NDP domain with an initial state. In NDP, initial states are defined in a *planning problem*:

Definition 4 (planning problem) A *nondeterministic planning problem* is a tuple <D; I; G>, where:
D = <P; S; A; T> is a nondeterministic planning domain.
I$\in$S is the initial state.
G$\in$S is the goal state.

Combining with the goal, a STS will lead to a planning problem, and we can generate a plan:

Definition 5 (plan) A *plan* for a planning problem < D; I; G> is a set of state-action pair <s, a >, where:
s is a state;
a is an action that can be applicable in s.

And there is at most one action for one state.

(Pistore *et al.*, 2005) has shown that a BPEL4WS process can be translated into a STS. From the algorithm and examples in last section, we can see that any state must have one and only one action after it, this is exactly a plan. The proof of the equivalence of plan and BPEL4WS process is the same with (Wu *et al.*, 2003) and (Pistore *et al.*, 2005).

Extracting State-Action-State Tuples

We can extract state-action pairs from the STS we build in last section as plan, but considering non-deterministic actions may have different effects, we have to extract STAT tuples (for LAM Module).

This algorithm is pretty naïve: we use *R* as STAT tuples, and for every element in *R* $\subseteq$ STS we use the first state and the action as a state-action pair, all these pairs consist a plan[13] .

Example 3: In Example 2, we extract plans as follows (We use S_0 for initial state and S_G for goal state from here):
Plan = { {S_0, Assign Action 1}, { S_1, BookRoom}, { S_2, Assign Action 2}, { S_2', Exit}, }
STAT = {{S_0, Assign Action 1, S_1}, { S_1, BookRoom, S_2}, { S_1, BookRoom, S_2'},{ S_2, Assign Action 2, S_G}, { S_2', Exit, S_G} }.

Extracting Action Models

In the last two sections, we discuss how to transfer a BPEL4WS program into a STS, and we address the problem of extracting STAT. In this section we will finally provide solutions for extracting action models from STAT.

LAM Module Architecture

LAM module has three components[14] : it first convert all STAT instances into its schemas, and then extract preconditions from all STAT schemas, and end with the extraction of the effects. The Instance2Schema component uses the same idea and method in (Yang *et al.*, 2007), and this conversion might be necessary for all LAM works[15] .

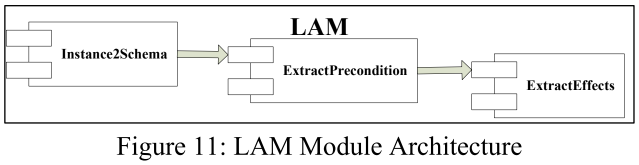


Figure 11: LAM Module Architecture

[13]This plan will not contribute to action model learning, but this has never been done before, and it might be useful in occasions like WSC solution verification as in (Foster et al., 2003).

[14]Another component which is in charge of extracting the condition effects is under construction.

[15]We refer readers to (Yang *et al.*, 2007) for more details. Simply put, it replacing all constants by their corresponding variable types.

Preconditions Extraction

The extraction of preconditions is an inductive learning method. We check every state before the action in all STAT schemas: if a proposition p belongs to every one of them then it is a part of preconditions.

Example 4: In Example 1, we finally have STAT about BookRoom action as:

STAT = {{ S_{A1}, BookRoom, S_{A2}}, { S_A, BookRoom, S_{A2}'}, { S_{B1}, BookRoom, S_{B2}}, { S_B, BookRoom, S_{B2}'}}

Here we suppose S_{A*} is from WSC_A and S_{B*} is from WSC_B.

The states before BookRoom are:

S_{A1} = {CustomerName = true, BookedFlag = false, BookSuccess = false}

S_{B1} = {CustomerName = true, BookFail = false}

Then the precondition for BookRoom should be:

Precondition = { CustomerName = true}.

Effects Extraction

We use a Case Based Reasoning (CBR) way for effects extraction. Before we move on, some terms should be introduced.

Definition 6: (**Static Assumption**) There are *no exogenous events* in a system, so no changes happen to the states except those performed by the controller.

This assumption is a basic assumption for classical planning (Ghallab *et al.*, 2004). Like most works in planning community, our methods are based on this assumption.

To further tell the effects of an action apart from the context, we need a definition for the changes.

Definition 7: (**CPS**) Suppose *p* is a proposition, *Prior* is the set of propositions for the state before action in STAT and *Tail* is the set of propositions for the state after action in STAT. If $p \in Tail$ and $p \notin Prior$, than *p* is a *changing proposition (CP)*. All changing propositions consist of *Changing Proposition Set (CPS)*.

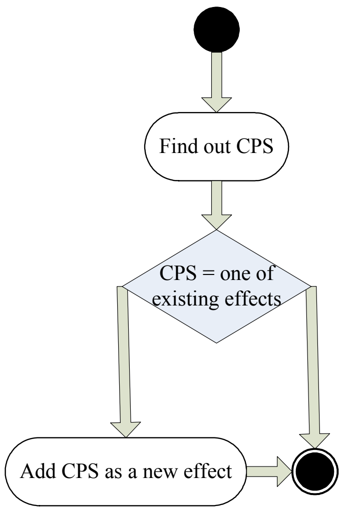


Figure 12 : Category Algorithm Flow Chart

For every STAT[16] , we run the Category Algorithm, see Figure 12.

Example 5: In Example 1, there are two STAT (ignoring the pair states):

STAT = {{ S_{A1}, BookRoom, S_{A2}}, { S_{B1}, BookRoom, S_{B2}}}

The CPS for STAT are :

CPS_A = { BookSuccess = true }

CPS_B = { BookFail = true }

And the effects are:

Effect1 = { BookSuccess = true }

Effect2 = { BookFail = true }.

This can be converted into extensions of PDDL like NADL+, NPDDL, and PPDDL. For example, it can be like the following in NPDDL:

Example 6: The result of Example 5 can be (not being included in our work yet) converted into NPDDL format as:

```
(:action action_name
:precondition (and (CustomerName = true))
:effect (and
(oneof (BookSuccess = true) (BookFail = true)))
```

Obviously, for BookRoom service in Example 1 BookSuccess and BookFail are mutex flags, but we can only know they are two possible out comings. In fact, without any human or semantic tools help, we can never tell what the relations among these effects are. This is one of the two reasons (the other will be stated at the end of this section) why our action models can be used for only the input domain. But if we can use these models as partial model for further learning, these action models can be used in similar domains.

Conditional Effects

Learning conditional effects component of LAM module is under construction. As far as we know in previous works only (Shahaf *et al.*, 2006) handles a very limited form of this problem. When moving to the non-deterministic field, learning conditional effects becomes more complicated as we have to tell conditional effects and non-deterministic effects apart.

Correctness

After the whole process, we now address the problem of the correctness of LAM.

Definition 8 (LAM Fairness Assumption): A set of LAM samples is *fair* if they satisfy the following requirements, here we suppose *p* is a proposition, *Pre* is the precondition of the action and *Eff* is one of the effects:

1. If $p \notin Pre$ than there exists a STAT, where *s* is the state before action and $p \notin s$.
2. For every *Eff*, there exists a STAT, where *Eff* is the CPS of it.

According to our precondition extraction method, if there exists a STAT, where s is the state before action and

[16]Here we will not use the pair state that was created in SplitStates module in "From BPEL4WS to STS" Section, for it is made up to represent the non-deterministic nature of the action.

p $\notin$ s, then p must not belong to every schema. So p will never be included in the precondition set. Similarly, reasoning goes with the effect proposition set. So apparently, under *LAM Fairness Assumption* the result of ExtractPrecondition module will not include propositions other than the precondition, and all the effects will be categorized in ExtractEffects module. This means we need a large number of WSC solutions to build a reliable action model, and it is the other reason why our learned action models can be used for only the input domain.

Conclusion and Future Work

In this paper, we address the problem of learning non-deterministic action models for Web Services from existing WSC solutions which are BPEL4WS programs. To do so, we first provide methods to translate BPEL4WS programs into a planning domain (STS), and then we extract STAT from STS, and extract action models from STAT at last.

We first use BPEL2STS module to translate a BPEL4WS program into a STS and then extract STAT tuples from STS, finally all STAT tuples are given to LAM module, to extract action models. In BPEL2STS, we first collect all relevant propositions, and then find states and transitions in BPEL4WS programs. The result of BPEL2STS is STS, and STAT can be easily extracted from it. In LAM, we use the propositions which appear in every prior state of the action in STAT as precondition of the action. Finally, under *Static Assumption*, we use different CPS as different effects. According to the reasoning in last section, under *LAM Fairness Assumption*, LAM module will output right action modules (see Example 4 and Example 5).

We test our approach on samples provided by ActiveBPEL[17] , e.g. async_echo, loanApprovalProcess[18] (LAP for short), multi-start_receives, repeatUntil, validate, and While samples. As these samples are not designed for non-deterministic action learning, we have to give each of them a corresponding program (to use more than one possible out comings of services). Part of the test results are listed in Table 1.

Basic Sample	Number of Programs	Number of ND-actions	Basic Structure
BookRoom	2	1	If/While
LAP	2	1	If
repeatUntil	2	1	While

Table 1: Part of the test examples

As a difference with previous work in this direction, we first translate BPEL4WS Executable processes into a planning domain. This can be very helpful in two ways: Learning action models for Web Services, and verifying WSC solutions in BPEL4WS programs. As our major contributions, learning action models from existing WSC solutions could eventually relieve people from manually writing action models for services. Meanwhile, it makes planning tools available for WSC solution verification. The gap between industry standards and academic researches could be bridged by our approach. Moreover we further extend the scope of learning action model into NDP. The methods we present in this paper can be used in other circumstances where non-deterministic action models need to be extracted.

Our work has a lot of original ideas here. First, from the WSC aspect of view, unlike (McIlraith & Fadel 2002), (McDermott, 2002), (Wu *et al.*, 2003), (Martínez & Lespérance, 2004), (Vukovic & Robinson, 2004) and (Peer, 2004), we consider the new trend in Web Service, the BPEL4WS.[19] Second, from the BPEL4WS using and translation point of view, unlike (Foster *et al.*, 2003), (Fu *et al.*, 2004), (Traverso & Pistore, 2004) and (Pistore *et al.*, 2005), we first address the problem of translating BPEL4WS Executable process into a planning domain. Third, from the learning action model point of view, unlike (Wang, 1995), (Balac *et al.*, 2000), (Blythe *et al.*, 2001), (Pasula *et al.*, 2004), (Shahaf *et al.*, 2006) and (Yang *et al.*, 2007), we first consider non-deterministic actions.

But this by no means is the end of the story. First, at current stage, we can only handle a subset of BPEL2.0, and it is a huge challenge for us to fully cover it. Second, the conditional effects learning module is pretty hard but necessary. Third, motivated by (Peer, 2004) and (Lerman *et al.*, 2006), we are trying to get models from WSDL files and learn action models based on these partial models. If we can get this done, we can also use learned models as partial models in similar domains. Finally if we could extend our work into partial observable environment, it would be more useful because most industry solutions are not complete, some even not sound.

Acknowledgements

This research is funded by Chinese Natural Science Foundation. The authors gratefully acknowledge the early work and discussions of Doctor K.H. Wu, and the helpful comments from the reviewers.

References

Balac, N.; Gaines, D.M.; and Fisher, D. 2000. Using Regression Trees to Learn Action Models. *IEEE Systems, Man and Cybernetics Conference*, pp. 3378-3383. Nashville, Octoboer, 2000.

Bertoli, P.; Cimatti, A.; Dal Lago, U.; and Pistore, M. 2003. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. *In Proceedings of ICAPS 2003 Workshop on PDDL, Informal Proceedings*, pp. 15-24.

[17]http://www.activebpel.org/samples/samples-3/samples.php.

[18]It is the sample used by (Fu *et al.*, 2004), too.

[19]Unlike (Wu, *et al.*, 2003) and (Kuter, *et al.*, 2004), we assume semantic descriptions for services are unavailable.

Blythe, J.; Kim, J.; Ramachandran, S.; and Gil, Y. 2001. An integrated environment for knowledge acquisition, *In Proceedings of the 2001 International Conference on Intelligent User Interfaces (IUI2001)*, Santa Fe, NM, 2001, pp. 13–20

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking, *Artificial Intelligence, 147(1/2),* pp. 35--84, (2003).

Foster, H.; Uchitel, S.; Magee, J.; and Kramer, J. 2003. Model-based verification of web service compositions. *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003).* pp. 152-161.

Fu, X.; Bultan, T.; amd Su, J.W. 2004. Analysis of interacting BPEL Web services. *In Proceedings of the 13th International Conference on World Wide Web (WWW 2004)*, pp. 221-230. New York, USA

Ghallab, M.; Nau, D.; and Traverso, P. *Automated Planning: Theory and Practice*. Morgan Kaufmann, May 2004.

IBM, Microsoft, and BEA. 2002. Web Services Business Process Execution Language Version 1.0

Jensen, R.; and Veloso, M. M. 2000. Obdd-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, *13*. pp. 189-226.

Kuter, U.; Sirin, E.; Nau, D. S.; Parsia, B.; and Hendler, J. A. 2004. Information Gathering During Planning for Web Service Composition. *In Proceedings of International Semantic Web Conference 2004*. pp. 335-349

Lerman, K.; Plangprasopchok, A.; and Knoblock, C. A. 2006. Automatically labeling the inputs and outputs of web services. *In Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006)*, Boston, MA, July 2006

Martínez, E.; and Lespérance, Y. 2004. Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning. *In Proceedings of the ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services*, pp. 62-69, Whistler, BC, June 3-7, 2004

McDermott, D. 2002. Estimated-regression planning for interactions with web services. *In Proceedings. of the AI Planning Systems Conference 2002*. pp. 204-211. AAAI.

McIlraith, S.; and Fadel, R. 2002. Planning with Complex Actions. *In Proceedings of International Workshop on Non-Monotonic Reasoning (NMR2002)* pp.356-364,April, 2002.

OASIS. 2007. Web Services Business Process Execution Language Version 2.0 (Primer)

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2004. Learning probabilistic relational planning rules. *In Proceedings of ICAPS 2004.* pp. 73-82.

Peer, J. 2004. A PDDL based Tool for Automatic Web Service Composition. *In Proceedings of the Second Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2004) at the 20th International Conference on Logic Programming*, St. Malo, France, LNCS 3208. Springer Verlag. pp. 149-163.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning in Asynchronous Domains. *In Proceedings of ICAPS 2005*. pp.2-11

Sandewall, E.; and Ronnquist, R. 1986. A representation of action structures. *In Proceedings of the 5th (US) National Conference on Artificial Intelligence (AAAI-86)*. pp. 89-97, Philadelphia, PA, USA, August 1986. American Association for Artificial Intelligence, Morgan Kaufmann.

Shahaf, D.; Chang, A.; and Amir, E. 2006. Learning partially observable action models: Efficient algorithms, *In Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006)*, Boston, MA, July 2006

Srivastava, B.; and Koehler, J. 2003. Web Service Composition - Current Solutions and Open Problems. *In Proceedings of ICAPS 2003 Workshop on Planning for Web Services*, June, Trento, Italy

Traverso, P.; and Pistore, M. 2004. Automated composition of semantic Web services into executable processes. *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, pp.7～11.

Vukovic, M.; and Robinson, P. 2004. Adaptive Planning Based Web Service Composition for Context Awareness. *In Advances in Pervasive Computing*, volume 176, pp. 247–. 252, 2004.

W3C. 2002. Web Service Description Language (WSDL) version 1.2

Wang, X. 1995. Learning by observation and practice: an incremental approach for planning operator acquisition. *In Proceedings of ICML-95*, pp.549-557. MK.

Wu, D.; Sirin, E.; Hendler, J.; Nau, D.; and Parsia, B. 2003. Automatic Web Services Composition Using SHOP2. *In Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, May 2003.

Yang, Q.; Wu, K.H.; and Jiang Y.F. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence, 171*.pp.107-143

Younes, H. L. S.; and Littman, M. L. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. *Tech. rep. CMU-CS-04-167*, Carnegie Mellon University, Pittsburgh, PA.