Learning Task Specific Web Services Compositions with Loops and Conditional Branches from Example Executions

Harini Veeraraghavan\(^1\), Roman Vaculín\(^2\), Manuela Veloso\(^3\)

\(^1\) General Electric Research (veerarag@ge.com), \(^2\) Institute of Computer Science, Academy of Sciences of the Czech Republic (vaculin@cs.cas.cz), \(^3\) Computer Science Department, Carnegie Mellon University (veloso@cmu.edu)

Abstract

Majority of the existing approaches to service composition, including the widely popular planning based techniques, are not able to automatically compose practical workflows that include complex repetitive behaviors (loops), taking into account possibility of failures and non-determinism of web service execution results. In this work, we present a learning based approach for composing task specific workflows. We present an approach for learning task specific web service compositions from a very small number of observations (one or more) of example service execution sequences (traces) that solve a given goal. The workflows learned by this approach generalize to the tasks justified by the observed execution trace. The generalization captures the repetitive executions of service sequences, conditional branching executions, and repetitions and branching resulting from failures. We evaluate the approach on a complex web services application involving arbitrary number of repetitive executions and failed executions.

1 Introduction

Automatic composition of web services is a challenging problem. While planning based approaches [16, 11, 8] are commonly used techniques for composing web services, such approaches have many drawbacks. The main conceptual challenges result from the fact that, in order to maximise the reuse potential, individual web services (WS) are typically defined in a task independent manner with only general applicability preconditions and effects. As an outcome, theoretically, each service could be employed in a very wide variety of scenarios. However, in reality when the specific task and its context is considered, despite their general applicability such services will often turn out completely useless since they were developed for a different purpose. Next, the execution of typical web services have non-deterministic results, and in many applications it is necessary to generate the workflows involving complex loops and branches, both of which is extremely challenging for planning based techniques. Finally, web services composition needs to be typically performed in open environments such as the Internet where the assumptions of a closed world knowledge employed by classical planning-based algorithms are no longer valid.

In this work, we approach the problem of automatic web services composition very differently – we suggest to generate the service compositions through learning from a very small number of example demonstrations. Instead of resorting to planning, we propose to learn a generalization of the workflow from an example of service executions that demonstrate how to solve a given problem (or achieve a given goal) by means of known individual services. Such example service execution demonstrations can be provided for example by a domain expert who knows how to solve problems in a particular domain. Our approach is motivated by the following observations. Firstly, most service compositions are specific for a given application context, rather than being generic. Second, it is important to realize that good, working compositions are based on applying some form of expert / domain knowledge. It is somewhat unrealistic to assume that even a very good planner (without any domain specific knowledge) can come up with high quality service compositions. Thirdly, very often good solutions are based on some sort of generalization of the already known solutions. In the context of web service workflows, such generalization can be based for example on substitution or parametrization of entities (e.g., services or constants) in example executions that are known to work. Finally, we assume that it is much easier to generate or elicit example service executions solving a given problem than to specify the overall complex generalized workflow.

The contribution of this paper is a demonstration-based learning approach to task specific web services composition. We present an approach wherein an example execution trace consisting of web services steps is generalized to
a task specific plan (workflow). We use the execution trace as an example plan, and we identify the set of data and other observable dependencies between the web services steps in the observed trace to produce a generalized plan that is able to solve similar problems. The generalized workflow respects the service ordering constraints that are supported (necessitated) by the identified dependencies, and at the same time it does not impose the ordering constraints where no supporting evidence can be deduced from the execution trace. The ordering of web services is captured as task specific preconditions for each web service. Thus the generalized workflow guarantees that a new problem will be solved in the same fashion as in the example trace, while keeping the flexibility as high as possible. In addition to learning the orderings of web services, our approach learns generalized workflows involving sequential repetitions (loops), conditional branches (e.g., including those resulting from failed execution of web services), and conditional branches that might be nested inside the loop.

Our approach to learning task specific plans is somewhat similar to the work of Winner and Veloso [15], however there are substantial differences. First, our work is applicable to the execution traces resulting from web services (WS) which are much more general than typical planning-based operators (typically based on PDDL operator definitions). In the WS domain services are defined by means of their typed inputs and outputs, and often also preconditions and effects. Furthermore, realistic WS operate with complex datatypes, such as complex records of data, classes, or containers (e.g., lists) of simpler datatypes. Classical plans learning does not address this problem, while we specifically address the problem of complex datatypes. Secondly, we handle more complex workflows such as conditionally branching executions occurring inside loops, whereas the work in [15] handles branches exclusively outside loops. Finally, we handle executions which can contain failures or executions that result in failures, while the work in [15] does not address this problem.

We organize the paper as follows. We first discuss the related work in Section 2. Then we introduce the the preliminaries and the background of our learning approach in Section 3. In Section 4 we present details of our approach to learning workflow generalizations as task specific plans, and finally present results and discussion in Section 5, followed by conclusions in Section 6.

2 Related Work

There are three areas of research related to our work, namely the classical planning based services composition approaches, the learning by demonstration techniques, and the research in the area of workflow and process mining.

In the area of “classical” approaches (i.e., those using no learning techniques) to service composition a wide variety of techniques have been studied. For instance, classical planning methods have been applied to WS composition [8], as well as HTN planning methods [16, 11], addressing composition of WSDL services and semantic web services as well. Majority of the planning-based approaches assume the availability of task specific preconditions in the web service descriptions [10], or plan/goal update rules, or make specific assumptions about the nature of the composite web service such as that there are no data-dependencies of the constituent web services [3]. However, as pointed out in [12], WS applications are developed by heterogeneous organizations which precludes the specification of task-specific preconditions such as step ordering preconditions. We address this problem by learning complex task-specific plans with data-dependencies as well as repetitions using WS without task-specific preconditions – the preconditions are learned as an outcome of our algorithms.

Our approach is based on learning by demonstration. Learning by demonstration has previously been applied successfully for example in robotics [9]. Similarly, the work in [5] presents an approach to learning HTN plans from human demonstrations with annotated sub-tasks, while the work in [6] presents an approach to learning looping plans where the beginning and end of loops are marked. Other approaches to learning plans with loops include the works in [7] using situational calculus and [4] using model checking. Unlike these works, our work does not require complete domain definitions, deterministic actions, loop annotations, and it handles domains with complex datatypes.

Finally, learning workflow or process models from execution traces is the subject of the process and workflow mining. Aalst et. al. [1] present a survey of process mining techniques. In process mining one crucial assumption is that enough data in the form of many execution traces or logs is available. Naturally, this allows mostly statistical techniques to be used for learning the structure of workflows. Such techniques cannot be used in our case, where we explicitly assume that only a very small number of execution traces is provided.

3 Preliminaries

The goal of learning workflows is to extract a generalized plan for the specific task such that similar tasks as the demonstrated task can be executed automatically without requiring computationally expensive search and planning. Similar tasks mean problems that achieve goals that unify with the demonstrated task and can be accomplished using the same ordering of steps and the same if not all operators as in the learned workflow. We assume that the demonstrated trace is the best possible way to accomplish the goal even if some steps in the trace may produce failed execu-
3.1 Structure of Workflows

A workflow typically consists of a sequence of partially ordered operators or web service steps. A partially ordered sequence is illustrated in Fig. 1. Partial ordering of steps means that every step in the sequence does not need to have a strict ordering constraint with respect to the previously executed or subsequently following steps. Although the steps $opA(x)$ and $opB(x)$ can be executed in any order, both steps must be executed to produce the preconditions for step $opC(x)$ whose effect achieves the goal.

The sequence in a workflow may also consist of conditional branches, where specific sub-sequences of steps are executed when certain conditions hold true. Fig. 2 illustrates such an example. In this case, the operator $opB(x)$ produces a non-deterministic effect. The example shown in Fig. 1 produces an effect $(b1 \ x)$ whereas it Fig. 2 it produces an effect $(b2 \ x)$ as a result of which a step $opD(x)$ executes to produce the precondition $(b1 \ x)$ for the execution of $opC(x)$.

However, workflows can be more complex and include repetition of a sequence of steps such as shown in Fig. 3 and Fig. 4. We call such repetition of behaviors (sequence of steps) as loops. We distinguish two different kind of loops: total body matched loops in Fig. 3 defined as follows:

1. **Total Body Matched Loops:** In a total body matched loop, all the repeating sequences of steps are identical in the steps as well as the order of steps used. Two steps or actions $opA(x1)$ and $opA(x2)$ are said to match when they unify in their names, argument types and in the set of preconditions and effects. Generalizing to a sequence, two sequences of steps constitute a loop when all the steps in the individual sequences unify.

2. **Relaxed Body Matched Loops:** In a relaxed body matched loop, only a sub-sequence of steps in all the instances of the loop must match completely in both the steps and their execution order. However, there must be at least two loop instances where the entire subsequence of steps in the loop match as in a total body matched loop (as demonstrated in Fig. 4).

In this work, we detect both kind of loops. The loops in question have two interesting properties:

1. There are no causal relationships between any of the steps across the different instances of the loop. In other words the repeating behaviors execute independent of each other.

2. All the steps in a sequence forming the instance of the loop execute in relation to a specific constant. The constants are $x1$, $x2$ in Fig. 3 and $x1$, $x2$, $x3$, $x4$ in Fig. 4. We refer to these constants as loop variable or terminal constants. The terminal constants are important for defining a loop. This is because if a particular step can execute without the terminal constant and produce the outputs relevant to all instances of the loop, by first property, it could not belong to the loop.

3.2 Web Services Execution Traces

In general, the workflows for the execution of the web services follow the basic concepts of sequences and loops as discussed in the former subsection. However, the web services steps themselves are far more complex. As such the execution of a web service can produce complex types of outputs including lists and records. The result of having such complex output types makes recovering the correct partial ordering of the steps and eventually learning the workflow structure very challenging. To illustrate the difficulty, let's take the example of a simple web services trace...
Figure 4. Example of a relaxed body matched loop. Only sub-sequences of steps in all instances of the loop match.

Figure 5. An example trace with list of outputs, and non-obvious repetitions from a book buying domain.

as depicted in Fig. 5. The example in Fig. 5 depicts a trace executed for buying books from web services provider such as the Amazon. In this example, the goal of the services execution is to buy different books on the topic "Bayesian Learning". As shown in the example, the first web service step produces a complex type of output consisting of a list of two different books and information related to those books.

This trace presents two different problems for learning the correct workflow. From the first look at the services, it would seem that the looping variable hereby referred to as the terminal type is the type BookTitle, as this is the input type on which two matching web services steps BuyBook are executed. However, learning the workflow based on the BookTitle type would result in a completely wrong generalization of the workflow such as depicted in workflow in Fig. 6a. This is because, the learned generalization will simply buy every single book item with a unique title returned by the SearchBooks service. This can result in unsuccessful executions as well as non-achievement of goals where there are two books by different authors but the same title. In fact, the correct terminal type for this problem is the type BookID. It is important to reveal the terminal type for the learning algorithm to generalize the correct workflow. The second problem results from the single execution of the web service LookupSeller. In this example, as the seller is the same for both the books B1001, B1002, the user simply reuses the result of one web services execution for buying both the books. However, both the books having the same seller is a coincidence. The resulting workflow where the seller information is looked up for only the first book can in fact result in an exception. To overcome the two problems, we introduce the notion of "context" of execution.

The execution context is based on the idea that the data in the individual entry of a list have no association with the data in other entries of the list. In other words, the individual tuples in the output list of the services define the execution context. Thus the context of execution of a web service step is defined as the set of facts associating its inputs to the constants that occur in the tuple of the output list of the web service steps that ensure those inputs. A web service step’s execution context can be in relation to multiple other contexts. In the example trace in Fig. 5, the execution context of the web service LookupSeller is associated with two different output tuples having the terminal constants B1001 and B1002. Clearly, as multiple services execute and produce lists of data, this in turn generates more contexts. However, those output contexts will form the sub-context of the same web service’s execution context. Once the context of execution is known, the terminal constants for each context can be detected. This results from the fact that all the steps that constitute a loop execute in relation to the terminal constant. Hence, every step must have an input or an output constant that is related to the terminal constant. By chaining through the set of facts that relates the inputs of every step in a subsequence of web service steps that constitute an instance of a loop, we can determine the terminal constant.

Fig. 7 depicts the representation of the web services using their execution context. As can be seen, now the execution steps are unrolled to reveal a loop structure for both the LookupSeller and the BuyBook service. Also using the context of execution of each service, we can also obtain the set of facts relevant to its execution with respect to the terminal constant as depicted in the Fig. 7. Analyzing this annotated
trace, helps to obtain the correct generalization of the workflow as shown in Fig. 6b.

4 Learning Task Specific Workflows from Observation of Web Services Execution

We define a web service as $W = \{I, P, O, E\}$, where $I$ refers to the set of inputs, $P$ the set of preconditions, $O$ the outputs, and $E$ the effects. The inputs are represented as a set of typed constants. The preconditions and effects are represented as conjunctions of predicates. The outputs of the web services may be individually typed outputs or a list $L = \langle T_1, T_2, \ldots \rangle$ with tuples of typed output.

The web services may use types that follow a hierarchical structure in the domain. Since we assume no access to the domain ontology, reasoning about types is constrained to identifying data types equivalencies. Our approach to workflow learning consists of the following two steps:

1. Preprocessing: Transforming the web services trace into annotated task specific web services trace.
2. Learning a generalized workflow from the observed annotated plan. To do that, the partial ordering of the steps in the trace are analysed to detect the partial ordering and data-flow constraints between the various steps to produce an annotated partial ordering graph.

4.1 Transforming the WS trace into an annotated task specific WS trace

The goal of preprocessing is to produce a transformed trace in a form that satisfies the requirements for loop and sequence detection as outlined in Section 3. First, any data type mismatches between the web service trace steps resulting from hierarchical data types are removed to facilitate the extraction of partial ordering constraints and data flow between the steps (Subsection 4.1.1). Second, the individual steps are annotated with their execution context (Subsection 4.1.2). This step enables the preprocessing to transform the annotated steps without complex lists of outputs. Third, any execution failures are handled by annotating the failures as effects (Subsection 4.1.3) of the transformed task specific service steps.

Central to the preprocessing is an Information State, referred to as IS. The IS can be viewed as the evolving knowledge base for the demonstration. The IS evolves with the execution of each web service step. The IS maintains: (1) Data type equivalences. (2) Constants and facts added as a result of execution of the individual web service steps. (3) Observed execution contexts of web service steps.

4.1.1 Detecting Equivalent Data Types

To detect the data type equivalences, the input constants for a web service step are matched with the constants existing in the IS in every context. Two data types that are not equivalent do not share the same constants. Hence, matching constants with mismatched types can only indicate that the types are equivalent. Thus upon detecting a mismatch between two matched types, their type equivalence is recorded in the IS. Denoting constants as $C_1$ and $C_2$ and their types as $T_1$ and $T_2$, we extract the equivalence as follows: $C_1 = C_2 \land T_1 \neq T_2 \implies T_1 \equiv T_2$.

Fig. 8 illustrates an example execution of two different steps used in a travel scheduling domain. As shown, the data types $\text{Orig}$, $\text{Dest}$ of the outputs produced as a result of the first service $\text{TripRequirements}$ execution do not match with the input data type $\text{Loc}$ of the service $\text{FindFlight}$ that uses the data $\text{Boston}$, $\text{NewYork}$ produced as a result of the first web service. Also shown is the current snapshot of the IS after the execution of the web service step $\text{FindFlight}$, where the data type equivalence has been recorded by the algorithm.

4.1.2 Execution Context Filtering

As previously defined, the context corresponds to the individual tuples of outputs of one or more web service steps that are assigned to the inputs of the current web service step in the execution trace. A context is created corresponding to each tuple of the list generated as outputs of web services execution. To detect the execution context of a web service, its input constants are matched with the constants in every context existing in the IS. This search may result in one or more matching contexts for the web service
Figure 8. Example snapshot of the information state IS after the execution of the step FindFlight

step. Upon detecting a matching context, the executed step along with its outputs is added to all its matched contexts as shown in the IS. Fig. 8. In the example shown in Fig. 8, the step FindFlight produces a list output with two tuples. The outputs are added to the context that matches its input list. Additionally two separate sub-contexts are created for the individual outputs. Additionally, the input constants are chained through the facts present in the matched contexts to identify the terminal constant for the step. In this example, the terminal constant is s123 and of type ID.

4.1.3 Annotating Execution Failures

Web services execution are often non-deterministic and may result in failures. Note that failures are easy to detect (e.g., in WSDL there are special messages for failures). Upon a failure, a web service typically returns no outputs. We assume, that after a service failure a recovery service call will follow. However, with the absence of effects that support the input data, there is no evidence that a plan ordering algorithm can use to infer the implicit data dependency between the failed services and the potential recovery services that follows it. To address this issue, upon detecting a failure of web service step, the preprocessing introduces an annotation to the task specific web service step’s effect that captures the fact that the failure occurred on the service when executing on the specific input data. Additionally, the preprocessing detects the web service step that are added as recovery steps that exhibit the same properties to the failed service. In this work, we assume there is only one service that acts as the recovery step and is executed in the same execution context as the failed step. Second, the recovery step will use the same input constants as the failed service’s constants. Third, the recovery step will produce the effects or output types that will otherwise be produced by the failed step. Using the service descriptions, the outputs and effects of two steps can be matched to identify that services produce equivalent effects. In reality, there can be a sequence of recovery steps that follow a failed execution, with possibly very different parameters. Recovery step detection is in itself a separate problem and is not in the scope of this work.

4.2 Learning Task Specific Looping Plans from Example Execution Trace

The next step in learning looping plans is to extract the rationale of the demonstrated plan in the form of partial ordering constraints of the step based on its preconditions and the steps that satisfy those preconditions. For this purpose, we make use of the SPRAWL algorithm [14].

In the last stage of learning, the partial step orderings of the demonstration is generalized into a task specific plan. For this purpose we use an approach as in our previous work applied to a robotics domain as described in [13]. Essentially, we learn the workflows such as described in the Section 3 that include sequences, branching sequences, sequential non-nested loops, and sequential loops with conditional branches. We do not detect loops with cyclic dependencies or loops nested with other loops. The algorithm has the following five steps:

1. The algorithm first detects sequences by using the partial ordering of the steps using the annotated partial order graph.
2. In the next step we detect conditional branches between steps. To do this, the algorithm first matches the subsequence of steps obtained from step 1 and then identifies a pair of steps that unify and have mutually exclusive effects. Then a branch is created with the effect of the steps as the condition and the sequence of steps that use that condition go in the body of the branch.
3. Loops are detected based on the properties of loops as described in Section 3. We first start by finding at least two instances of same steps (steps that unify) and execute without any dependencies. Next, we expand the search to both before and after the step executions to add steps that exhibit the same properties to the subsequence. The result of this exhaustive search procedure are longest common subsequences of steps or loops. However, note that the complexity of this part of the algorithm is quadratic in the number of steps in the trace.
4. In the next step, the algorithm merges all such longest common subsequences of steps. For those subsequences that completely match, they are simply merged into one loop. However, there may be subsequences where only portion of the subsequences match completely for the two compared loops. Using the subsequence of steps that unify, the algorithm proceeds to detect steps that produce non-deterministic effects. The branches are then created similar to step 2 and the generalized branches are added to the loop. The result of this process are all unified loops
detected in the plan.  
5. Finally all the generalized steps are sequenced by their partial order. Steps that have no ordering constraint are left unordered.

5 Results and Discussion

We tested our learning approach on web services traces generated by a human user using a graphical software tool invoking real web services from the domain of scheduling personnel evacuation [2]. The web services steps had all the complexities as described in the preceding sections such as data type mismatches between steps, complex data types of outputs, conditionally executing sequences, and failed exception of the steps. The example traces make use of a number of steps to obtain travel reservation for each person. The goal of the learning algorithm was to learn a correct workflow that automates the process of making travel reservations for an arbitrarily large number of people with a variety of requests using a sequence of steps.

Figure 9. Snapshot of web services from the experimental domain. The services are simplified with only a few effects shown to improve clarity.

We tested the efficacy of our approach on a number of problems that depicted two different classes of workflows. The first class included workflows involving sequential loops with no conditional branches inside the loops. The second class included workflows that had conditional branches inside the loops. Also the second class of examples incorporated step executions with failures. A snapshot of the services used in this application is shown in Fig. 9.

Fig. 10 shows an example plan learned from the observation of an execution trace as shown. In this trace, the user reuses the results of most of the services except for the patient specific services such as S6 (GetArrivalTime) and S9 (ReserveSeat). Our approach correctly identifies the context of all the remaining service step executions and identifies the correct looping workflow. As shown it also correctly reasons about the relevant predicates for the specific tasks’s execution and adds them to the precondition list of the services. Without these preconditions, the services are too general to execute and achieve the desired goal. For example, there are two different lookUpAirport steps executed, one in relation to the origin of the patient, and the second in relation to the destination of the patient. However, the web services only indicate the type restriction namely, LocationID. Furthermore, the type LocationID is used as an equivalent type for the AirportID in the LookupMission service.

Fig. 11 shows an example workflow learned from an observation of an execution trace consisting of over 50 steps. For the sake of clarity we do not show the trace. We verified the correctness of the workflow through manual inspection. Additionally, the workflows returned by our learning approach were used as a part of an automated integrated learning framework [2]. Our workflow or plan learning approach employs general principles and easily applies to a variety of domains. As a matter of fact, in a previous work, we have successfully employed our loops and conditional branch detection algorithms to a completely unrelated domain in robotics [13]. We have also tested our algorithms on problem sets with a very large number of patients and requirements. However, the approach will not generalize steps in the plan that are irrelevant. Irrelevant steps are those which have no apparent reason towards achieving the goal.

Figure 10. Task specific workflow learned from an example demonstration sequence. The example execution trace and the learned workflow are depicted.
Figure 11. Task specific workflow learned from an example demonstration sequence. The workflow depicts branches learned in the looping workflow.

What this means is that if the human user introduced some steps in the demonstration example that seemingly achieve a redundant effect that can be justified by a different service, such services will be dropped from the learned workflow. Hence, one consequence of our learning approach is that it also learns optimal workflows. By optimal, we mean workflows with no redundant steps and that are completely justified the training examples. Another feature and in some domains limitation of our approach is that the algorithms do not extract any ordering preference conditions for step execution. As a matter of fact, the loop learning algorithms will benefit if the ordering preference conditions for ordering the patients were not available. Otherwise, the entire plan will be recognized as an ordered sequence instead of a loop.

6 Conclusions

In this work, we presented a method for learning task specific workflows with loops and conditionally executing branches for web service composition from example demonstration. We introduced an approach for learning workflows from challenging web service traces where the web service steps may have data type mismatches, complex output data types, and non-deterministic executions, such as failed exceptions. Our approach learns the correct workflows that are justified by the demonstration to learn workflows with loops and conditionally branching workflows.

References