Workflow Inference: What to Do with One Example and No Semantics

Fusun Yaman and **Tim Oates**
Department of Computer Science & Electrical Engineering
University of Maryland Baltimore County
{fusun, oates}@cs.umbc.edu

Abstract

It is much easier to get human experts to provide example action sequences than declarative representations of either the semantics of the atomic actions in the sequences or the workflow used to generate the sequences. To address this particular instance of the knowledge acquisition bottleneck, this paper describes an algorithm called Workflow Inference from Traces (WIT) that learns workflows from a single action-sequence (trace) without the need for action semantics (i.e., preconditions or effects). WIT is based on model merging techniques borrowed from the grammar induction literature. It starts with a workflow that generates just the observed trace, generalizing with each merge. Prior to the merging phase, an alphabet of action types is created in which similar action instances are grouped according to their input/output characteristics in the trace. It is a sequence of tokens in this alphabet that is merged. We empirically evaluate the performance of WIT using a novel measure of similarity between workflows. This evaluation takes place in an instance of the well-known domain proposed by Berners-Lee, Hendler, and Lassila (Berners-Lee, Hendler, & Lassila 2001) in which actions correspond to accessing web services.

Introduction

It is much easier to get human experts to provide example action sequences than declarative representations of either the semantics of the atomic actions in the sequences (i.e., their preconditions and effects) or the workflow used to generate the sequences. Yet having such declarative representations can enable, among other things, tools for workflow automation. For example, a recent National Science Foundation workshop on the Challenges of Scientific Workflows concluded that "significant scientific advances are increasingly achieved through complex sets of computations and data analysis ... [that are] ... often represented as workflows of executable jobs and associated data flows", and that "domain scientists consider workflow as a crucial and underrepresented ingredient in Cyberinfrastructure". There is a need in this important domain, and a wide variety of others, for methods and allow workflows to be extracted from action sequences with minimal knowledge engineering.

To address this particular instance of the knowledge acquisition bottleneck, this paper describes an algorithm called Workflow Inference from Traces (WIT) that learns workflows from a single action sequence (trace) without the need for action semantics. WIT borrows techniques from the grammatical inference (GI) literature. In particular, it is based on model merging algorithms. WIT begins with a workflow that generates a single trace, the one observed. It then iteratively chooses states in the workflow to merge, thereby generalizing and producing a workflow that can generate increasingly varied traces.

Whereas the input to most GI algorithms is an alphabet (set of tokens) and a set of strings over the alphabet that are known to be positive (in the target language) or negative examples, the input to WIT is a sequence of actions that take inputs and produce outputs. WIT constructs sets of actions that are similar according to their input/output behavior in the trace, replaces each action in the trace with the identifier of its corresponding set, merges the resulting automaton, and extracts a workflow which includes the structure (a graph) and a set of constraints on inputs and outputs.

The WIT algorithm was tested on a web service domain first described in the seminal semantic web paper (Berners-Lee, Hendler, & Lassila 2001), which has been implemented and used by many other researchers. Results with a novel similarity metric for comparing workflows show that WIT performs well at capturing the target workflow.

The remainder of this paper is organized as follows. The next section defines the workflow inference problem formally and gives some of the terminology used in the paper. Following that the WIT algorithm is described, a workflow similarity metric is presented, and the empirical evaluation is described. Finally, we conclude and point to future work.

The Workflow Inference Problem

We assume the existence of three finite sets of symbols denoted $\mathcal{A}$, $\mathbf{D}$, and $\mathbf{N}$ that represent action names, data types, and variable names respectively. A variable definition is a tuple of the form $\langle x, d\rangle$ or $\langle x, [d]\rangle$ where $x \in \mathbf{N}$ and $d \in \mathbf{D}$. In the former case x is of type d, and in the latter case x is a set whose elements are of type d.

Next we define actions and action instances that are the building blocks of workflows and traces respectively.

An *action* is a triple $A = \langle a, I, O\rangle$ where $a \in \mathcal{A}$, and I

and O are sets variable declarations. Informally, an action is represented by an action name and a list of typed input (I) and output (O) parameters. We will use $Inputs(A)$ and $Outputs(A)$ to denote the set of variables in I and O, respectively, and $Action(A)$ to denote the type of action A, i.e., $Action(A) = a$. For example $getGeneric = \langle generic, \{\langle RxIn, Medicine\rangle\}, \{\langle RxOut, Medicine\rangle\}\rangle$ is an action that takes a medicine name as input and outputs the generic name of the input medicine.

Given a set of variables $X = \{x_1, \ldots x_n\}$ an assignment $M = \{(x_1, v_1) \ldots (x_n, v_n)\}$ for X maps every $x \in X$ to a value v.

An action call is triple $AC = \langle A, I, O\rangle$ where A is an action, I is an assignment for $Inputs(A)$, and O is an assignment for $Outputs(A)$. Intuitively, an action call is an action with an assignment to all of its input and output parameters. For example $\langle getGeneric, \{(RxIn, Advil)\}, \{(RxOut, Ibuprufen)\}\rangle$ is an action call where the action $getGeneric$ takes $Advil$ as input and outputs the generic name $Ibuprufen$.

A trace is a list of action calls $[ac_1, ac_2, \ldots, ac_n]$. The actions in the trace are totally ordered corresponding to the order in which they were observed.

A workflow defines valid sequences of action calls to achieve a certain task. It has two components, a graph and a set of input/output constraints. In the following definition we will assume the existence of two special actions, $start$ and $stop$, that have no inputs or outputs. We will use $start$ and $stop$ actions in a similar way to the start and end states of a finite state machine.

Definition 1 *Given a set of actions A containing the $start$ and $stop$ actions, a workflow W is a tuple $\langle G, C\rangle$ where G is a directed graph and C is a set of binding constraints such that*

- *Every node n in G is associated with an action $a_n \in A$, denoted $Action(n) = a_n$.*
- *There is a unique node i in G with 0 in-degree and $Action(i) = start$.*
- *There is a unique node f in G with 0 out-degree and $Action(f) = stop$.*
- *Every constraint $c \in C$ is of the form $n.i = n'.o$ where n and n' are nodes in G such that there is a path from n' to n and $i \in Inputs(a_n)$ and $o \in Outputs(a_{n'})$.*

Furthermore the nodes i and f are the start and stop node of W, denoted $start(W)$ and $stop(W)$, respectively.

Example 1 *In addition to the $getGeneric$ action described earlier, suppose we have the following action that orders the input medicine from an online pharmacy and returns the receipt as output : $order = \langle orderMedicine, \{\langle RxIn, Medicine\rangle, \langle pharmacy, StoreName\rangle\}, \{\langle receipt, ID\rangle\}\rangle$. Using these two actions we can define the workflows $\langle G_1, C_1\rangle$, $\langle G_2, C_2\rangle$ and $\langle G_3, C_3\rangle$ where G_i and C_i are shown in Figure 1.*

A workflow is consistent with a trace if it can generate the sequence of action calls in the trace. More formally:

Definition 2 *A workflow $W = \langle G, C\rangle$ is consistent with a trace $[ac_1, \ldots, ac_k]$ iff there is a path $[start(W), n_1, \ldots, n_k, stop(W)]$ in G such that for every ac_i, $Action(ac_i) = Action(n_i)$ and for every constraint $n_i.in = n.out$ in C there is a $j < i$ such that $n_j = n$ and $ac_i.in = ac_j.out$. Furthermore T is a characteristic trace for W if every edge in G is traversed in P.*

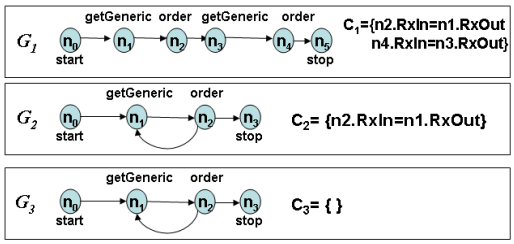

Figure 1: Example workflows to order one or more drugs in generic form.

Example 2 *The trace $[\langle getGeneric, \{(RxIn, Advil)\}, \{(RxOut, Ibuprufen)\}\rangle, \langle orderMedicine, \{(RxIn, Ibuprufen), (pharmacy, CVS)\}, \{(receipt, R01)\}\rangle]$ is consistent with the workflows $\langle G_2, C_2\rangle$ and $\langle G_3, C_3\rangle$ but not $\langle G_1, C_1\rangle$ shown in Figure 1.*

Given a trace T there are multiple workflows that are consistent with T, some more informative than the others. For example there is a trivial workflow that has a fully connected graph containing a node per action type and an empty set of binding constraints that can produce any trace. Another extreme workflow would be a chain of the same length as T that would produce just T. Intuitively we are interested in a minimal workflow which is as small as possible and has captured as much data flow as possible. The following definition formally presents the minimal workflow.

Definition 3 *A workflow $W_1 = \langle G_1, C_1\rangle$ is more specific then a workflow $W_2 = \langle G_2, C_2\rangle$ iff $|C_1|/Vars(W_1) > |C_2|/Vars(W_2)$ where $Var(W_1)$ and $Var(W_2)$ are the total number of inputs in actions of W_1 and W_2 respectively.*

Let $W = \langle G, C\rangle$ be a workflow and T be a characteristic trace for W. W is a minimal workflow w.r.t. T iff there is no workflow $W' = \langle G', C'\rangle$ such that 1) T is a characteristic trace for W' and 2) W' is more specific then W and 3) G' has less number of nodes then G

Example 3 *In Figure 1 , the workflow $\langle G_1, C_1\rangle$ and $\langle G_2, C_2\rangle$ are more specific then $\langle G_3, C_3\rangle$. Let T be a trace that is a repetition of the trace in Example 2 then $\langle G2, C_2\rangle$ is a minimal workflow w.r.t. T.*

We are now ready to define the workflow inference problem and its solution, which is finding the minimal workflow that is consistent with a given trace.

Definition 4 *A workflow inference problem (WIP) is a tuple $\langle AD, T\rangle$ where AD is a set of actions and T is a trace. A workflow W is a solution to $\langle AD, T\rangle$ if W is minimal w.r.t. T.*

Algorithm

Our Workflow Inference from Traces (WIP) algorithm borrows techniques from the field of grammar induction. Before describing the WIT algorithm, we summarize the differences between the workflow inference problem (WIT) and

the grammar induction (GI) problem. In general, the inputs to a GI algorithm (Oates, Desai, & Bhat 2002) are an alphabet (i.e., the set of terminals in the language of the target grammar) and a set of strings accepted by the target grammar. In some cases a set of negative examples, i.e., strings that are not in the language of the target grammar, is also provided. Instead of an alphabet, the input to a WIP algorithm has a set of action schemata, i.e., the names of the actions and a list of their inputs and outputs along with their types. Also the WIP is defined with respect to a single trace as opposed to a set of traces. Note that WIT does not require any information about action semantics, such as their preconditions or effects.

One general technique for solving GI problems when the target grammar is known to be regular is to create a deterministic finite state automaton (DFA) that accepts just the positive examples where the edges are labeled with terminals. The next step is to merge states with common prefixes while verifying that none of the negative examples are accepted by DFA after the merge. As a result, the final DFA is more general than the initial one and the grammar represented by the final DFA accepts more than just the positive examples. We cannot directly apply this technique to the WIP because there is no alphabet of constant terms. Second, since we have only one positive example our initial DFA is simply a chain, forcing us to use a different criterion to find mergable states. The WIT algorithm has 3 steps, alphabet generation, step generalization and workflow extraction. The following subsections explain each step in detail, and then we present the algorithm.

Alphabet Generation

The aim of this step is to group instances of the same action based on input bindings. Given a trace T, we identify for every input i of every action call c the producers, i.e., an action call p in T before c such that one of the outputs of p, say o, matches the value of i. Here matches is defined as:

- if o is a simple type then it is the same type as i and has the same value
- if o is a set then one of the elements in o matches i

In this case we denote the match as $c.i = p.o$ and we say that $p.o$ is a producer of $c.i$. If there is more than one producer for $c.i$ we select one of them based on a list of domain independent preference rules, e.g. prefer the producer that is closer to c in the trace[1]. Note that for some action calls some of the inputs won't have any producers. In that case, we just assume the value of that input is constant or given.

Next, we classify the action calls based on the similarity of their producers and action types. Two action calls a_1 and a_2 are *similar* if

- $Action(a_1)$ and $Action(a_2)$ are the same type of action A and for every input $i \in Inputs(A))$ the producers are the same, i.e., $a_1.i = p.o$ and $a_2.i = p.o$, or

[1]More complex selection preferences are encoded in WIT that can handle not only simple types but also composite types for which the matching criteria is extended to allow the fields of an object as a producer for a simple data type.

Action	Inputs	Outputs
getGeneric	RxIn=Advil	RxOut=Ibuprufen
order	RxIn=Ibuprufen pharmacy=CVS	receipt='R01'
getGeneric	RxIn=Tylenol	RxOut=Paracetamol
order	RxIn=Paracetamol pharmacy=CVS	receipt='R02'

Table 1: Action Calls in Example 4

- $Action(a_1)$ and $Action(a_2)$ are the same type of action A and for every input $i \in Inputs(A)$ the producers are similar, i.e., $a_1.i = p_1.o$ and $a_2.i = p_2.o$ where p_1 and p_2 are similar.

Finally we create a term for each action call in the trace such that similar action calls are associated with the same term. The the set of terms associated with the action calls is our alphabet. The term an action call a is mapped to is denoted by $term(a)$.

Example 4 *Suppose $T = [a_1, a_2, a_3, a_4]$ is a trace and a_i's are as given in Table 1. The only matches we have are $a_2.RxIn = a_1.RxOut$ and $a_4.RxIn = a_3.RxOut$. The first item in definition of similarity says that a_1 and a_3 are similar – because there is no producer for their inputs. Then by the second rule we find that a_2 and a_4 are similar because $a_2.RxIn$ and $a_4.RxIn$ have similar producers. Thus $\Lambda = \{g, o\}$ is the alphabet for this trace and $term(a_1) = g$, $term(a_2) = o$, $term(a_3) = g$, $term(a_4) = o$.*

Step Generalization and Workflow Extraction

Given the alphabet, we build a finite state machine F representing the input trace $T = [a_1 a_2 \dots a_k]$. F contains the states $s_0\ s_1\ s_2 \dots s_k$ and for every $0 < i \leq k$ there is a transition between s_{i-1} and s_i with the label $term(a_i)$. The start state and end state of F will be s_0 and s_k respectively. Next we iteratively merge the states s_i and s_j iff they both have an incoming edge with the same label. If any of the merged states is a final state then the resulting state will be a final state. The order of the merges does not matter. The intuition behind this is that actions with similar input bindings lead to similar states in a workflow. The resulting finite state machine (not necessarily deterministic) will contain exactly one sink node per term in the alphabet.

Given the trace T, the alphabet Λ, and the finite state machine F' produced at the previous step we can build a workflow $W = \langle G, C \rangle$. Note that in F' for any state all the incoming edges have the same label and every label corresponds to a set of action calls with the same action type (ensured by the similarity definition). Thus G will be very similar to F' with the main difference being edge labels will be removed and nodes will be associated with the action types corresponding to the incoming edge labels in F'. The binding constraints C will contain all the matches used for generating the alphabet but the action calls in a match expression will be replaced by the node they correspond to.

Example 5 *Let T be the trace and $\Lambda = \{g, o\}$ be the alphabet in the Example 4. Then step generalization will start with*

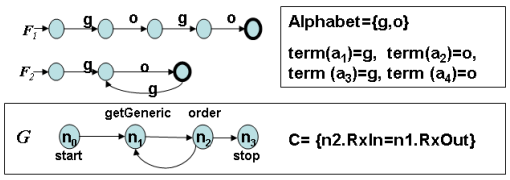

Figure 2: Step generalization and workflow extraction for the trace in Example 4

F_1 in Figure 2 and conclude with F_2. Then the workflow extracted will be $\langle G, C\rangle$ where G and C are as in Figure 2.

WIT algorithm

Algorithm 1 is the pseudo code of the WIT algorithm which presents the computation of the three stages explained above. Even though WIT is not guaranteed to find a minimal workflow, it always produces a workflow for which the input trace is characteristic.

Theorem 1 *Suppose A is a set of actions, T is trace, and $W = WIT(A, T)$ is a workflow produced by WIT. Then T is a characteristic trace for W.*

Algorithm 1 $WIT(A, T)$

Inputs: A is a set of actions, $T = \{a_1 \dots a_k\}$ is a trace
for all a where $a \in T$ **do**
  $Match(a) = \emptyset$
for all $a.i$ where $a \in T$ and $i \in Inputs(a)$ **do**
  Select a producer $p.o$ for $a.i$ and add to $Match(a)$
$\mathcal{M} = \{(a_1, t_1) \dots (a_k, t_k)\}$ where t_i=t_j if $Action(a_i)$=$Action(a_j)$ and $Match(a_i)$ = $Match(a_j)$
repeat
  if $\exists (a, t), (a', t') \in \mathcal{M}$ such that $t \neq t'$ and $\texttt{SIMILAR}(a, a', Match(a), Match(a'), \mathcal{M})$ is true **then**
    $\mathcal{M} = \mathcal{M} \cup \{(a', t)\} - \{(a', t')\}$
until $\mathcal{M}$ not changed
Let Λ be the set of all t such that $(a, t) \in \mathcal{M}$
Let F be an FSM with states $s_0 \dots s_k$
for all $0 < i \leq m$ **do**
  Let t be the symbol such that $(a_i, t) \in \mathcal{M}$
  Add a transition in F from state s_{i-1} to s_i with label t
while $\exists\ s_i, s_j$ in F with the same incoming transition **do**
  Merge s_i, s_j
Let G be a graph that has a node n_t for every $t \in \Lambda$ and $C = \emptyset$
for all nodes $n_t \in G$ **do**
  Arbitrarily pick an action call a such that $(a, t) \in \mathcal{M}$
  $C = C \cup \{n_t.in = n_{t'}.out | a_t.in = b.out \in Match(a) \wedge (b, t') \in \mathcal{M}\}$
for all Pairs of n_t and $n_{t'}$ **do**
  Let s_t and s'_t be the states in F with the incoming transitions labeled t and t'
  if There is an edge from s_t to s'_t **then**
    Add an arc from n_t and $n_{t'}$ in G
return The workflow $\langle G, C\rangle$

Comparing workflows

In this section we propose a novel similarity metric to compare two workflows. We will compare the workflows based on the languages they represent. The novelty of our approach comes form the fact that we can apply language comparison to workflows with data binding constraints. In the first step we compute the shared alphabet of the workflows based on binding constraints and node types. Then we represent the two workflows as FSMs utilizing the shared alphabet. However, since FSMs with cycles have infinite languages we cannot simply generate all possible strings and then compare the languages. Thus we need to approximate the language of an FSM. To this end we utilize 'n-grams' (Mahleko, Wombacher, & Fankhauser 2005) (i.e., sequences of length n leading to a given state) of the FSM because a combination of n-grams can be used to construct all possible execution sequences of a single automaton.

Computation of the shared alphabet of the two workflows is very similar to the alphabet generation step in WIT. The major difference being we don't have to guess the producers of the action inputs as they are already declared in the binding constraints of the workflows. We also need to rename the nodes so that the set of nodes in the workflows will be disjoint. We will say two nodes in any of the workflows are similar iff they have the same action types and they either have the same binding constraints or all the producers in their binding constraints are similar. Finally we assign a term to each node such that two nodes are assigned to the same symbol iff they are similar. Let this mapping be $\mathcal{M}$.

Given the mapping $\mathcal{M}$ from nodes of the workflows to the shared alphabet, the induced FSM F for a workflow $\langle G, C\rangle$ is defined as:1) For every node n in G there is a state s_n in F 2) There is an edge (s_n, s_m) in F iff there is an edge (n, m) in G 3) Every incoming edge to s_n is labeled as $\mathcal{M}(n)$.

The set of n-grams of a state s in an FSM is the transition sequences of length n leading into s. The set of n-grams of an FSM F is denoted by $Lang(F, n)$ and it contains the n-gram sets of every state in F.

Definition 5 *Let W_1 and W_2 be two workflows and F_1 and F_2 be their induced FSMs. Then $S(W_1, W_2, k)$ is the k-similarity of W_1 and W_2 and it is equal to $|Lang(F_1, k) \cap Lang(F_2, k)|/|Lang(F_1, k) \cup Lang(F_2, k)|$*

Experiments

In this section we empirically evaluate the performance of WIT using the measure of similarity between workflows. We use the medical scheduling planning domain used in (Sirin *et al.* 2004) and which is based on the scenario that was originally described in (Berners-Lee, Hendler, & Lassila 2001). The scenario is as follows: Pete and Lucy need to buy a list of prescriptions for their mother's treatment and then try to schedule appointments for a list of diagnostic tests to be performed. Since their mother is ill one of them will be driving her to the treatment center so they need to take into account their schedules as well as the hospitals and pharmacies that accept their mother's health insurance. All the actions in this domain are described as Web Services

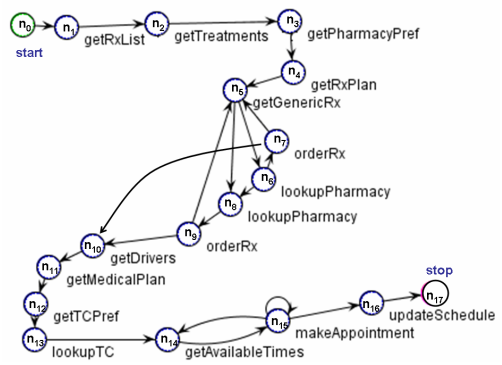

Figure 3: The target workflow in the experiments.

some of which gather information, e.g. learn available appointment times, and some alter the world, e.g. schedule an appointment.

Our test domain contains 14 actions. The graph in Figure 3 shows the acceptable sequence of actions for achieving the tasks Pete and Lucy have. Basically the workflow has the following steps: 1) Retrieve the list of medicines and tests, 2) Lookup for the pharmacy plan and the pharmacy preference filter (which is a radius), 3) For every medicine in the list lookup the generic medicine name 3.1) If there is a generic medicine available then lookup a pharmacy that is within the preferred distance, accepts the pharmacy plan and carries the generic medicine. If one exists then order the generic medicine, 3.2) Otherwise find a pharmacy that satisfies the same constraints but carries the original brand medicine and order the medicine 4) Get a list of available drivers (for this case it is just Lucy and Pete) 5) Get the medical plan info and treatment center preference 6) Lookup the treatment centers that offer the tests in the list and accepts the plan and are within the preferred distance 7) Try every treatment center for an appointment time for all treatments until one that does not conflict with one of the drivers is found 8) Update the designated driver's schedule with the appointments. For the rest of this section we will call the workflow consistent with these steps (and with figure 3) the α workflow. In a variation of the α workflow which we call the β workflow step 5 is be performed just after step 2.

We have performed two sets of experiments. In the first one α is the target workflow and we investigate the quality of the output workflow when the alphabet generation step correctly alphabetizes the input trace, i.e. choses the correct producers for every input. Note that, in any trace that is consistent with α, the correct producer of any input value is always the latest producer. The second set of experiments uses β as the target workflow. This time the latest produced value is not necessarily the correct one. For example if in a trace consistent with β the action calls $getTCpref$ and $getPharmacyPref$ output the same radius value then the producers of the $lookupPharmacy$ action calls will be $getTCpref$, which is not intended. In the second set of experiments we investigate how similar the WIT output and the β workflow when the generated alphabet is incorrect.

In our experiments we have randomly created the lists of prescriptions and medical tests as well as the available pharmacies and treatment centers. We have also ensured that at least one of the drivers and treatment centers have a common available time slot for the tests to be performed. With a randomly generated initial state, we traversed the target workflow $W^\star$ to create a trace consistent with $W^\star$. We ran WIT with the input traces and compared their similarity to the target workflows. Table 2 summarizes our results where α and β are the target workflows in the first and second rows. The first and second columns of Table 2 shows the minimum and maximum 2-similarity values between a WIT output and $W*$. We also compared the WIT output for a trace T with $Explored(W^\star, T)$, which is the part of $W^\star$ that is traversed to produce T. The third and forth columns of Table 2 shows the minimum and maximum 2-similarity values between a WIT output and $Explored(W^\star, T)$ for some T.

Target	$W^\star$		$Explored(W^\star, T)$	
	Min	Max	Min	Max
α	0.63	0.96	1	1
β	0.40	0.55	0.45	0.71

Table 2: Min and max 2-similarity for α and β workflows

Results for target α: WIT can find the workflow $Explored(W^\star, T)$ in all cases. Note that $Explored(W^\star, T)$ is also the minimal workflow. It is interesting to see that WIT can never exactly recover the target workflow, as the maximum 2-similarity value is 0.96. That is due to the fact that in any trace that is consistent with α only one incoming edge of node 10 will be explored, thus WIT has no evidence to produce to second edge. More generally if the target workflow has a node n with more than one in or out degree and there is no loop containing n then WIT can not produce the target workflow. This is a natural result of learning with a single example trace. Similarly the minimum 2-similarity is achieved when the input trace has no evidence of loops, i.e. there is just one medicine and the first treatment center has available appointments.

Results for target β: In this case as a result of wrong alphabet generation, WIT can not find the workflow $Explored(W^\star, T)$. 2-similarity with $Explored(W^\star, T)$ is minimum when $Explored(W^\star, T)$ has all the paths from n_5 to n_{10}. Naturally the maximum similarity is achieved when only one path from n_5 to n_{10} is in $Explored(W^\star, T)$. The 2-similarity value for β and WIT output is even lower since β contains all the nodes and have a larger 2-gram set then the 2-gram set of $Explored(W^\star, T)$.

Related Work

There is a large body of work in workflow/process mining (Weijters & van der Aalst 2001; Agrawal, Gunopulos, & Leymann 1998; Cook & Wolf 1998) in which the goal is to construct the process that produced an event log. Different representational schemes have been utilized to represent workflows such as petri-nets (Weijters & van der Aalst 2001) and finite state machines (Cook & Wolf 1998). Some of the existing work on workflow mining employs grammar inferencing techniques to find the target workflow. However in these work the events are uniquely determined by the

event name and they are not parametric, thus the alphabet is fixed and known. However WIT deals with different instances of the same action and extracts an alphabet based on the dataflow in the example trace.

Inductive program synthesis (Bauer 1979) and programming by demonstration (Lau & Weld 1999) are other related research areas that is closely related to our work. For once these works do not ignore the input output relationships. While most of them employ inductive logic programming to recover the target program given the example executions none applies the grammar inferencing techniques to generalize an example trace as WIT does. More over the example executions are annotated with the start and end of the loops and sometimes even the iterations of a loop is identified (Lau & Weld 1999).

Finally AI planning literature has some work on learning a domain specific planner from a plan trace (Winner & Veloso 2003; Nejati, Langley, & Konik 2006). Distill (Winner & Veloso 2003) generalizes the plan trace by using a goal directed approach and produces a domain specific planner that can solve new problems. Similarly, Nejati et. al (Nejati, Langley, & Konik 2006) applies a goal directed approach to extract macro actions in the form of HTN methods. Both works require the action semantics, i.e. preconditions and effects of the actions, but in return they produce the possible action sequences along with the branching conditions, i.e. when to take a certain branch in the workflow.

Several similarity metrics have been proposed in the literature. Some are based on comparing the languages the workflows represent (Mohri 2003; Mahleko, Wombacher, & Fankhauser 2005). The basic idea is to measure the distance between the strings produced by the workflows. In (Mahleko, Wombacher, & Fankhauser 2005) the use of n-grams to approximate the infinite language produced by a workflow has been proposed. Other similarity metrics ignore the language aspect and are based on comparing the structure (Chartrand, Kubicki, & Schultz 1998; Messmer & Bunke 1998; Du *et al.* 2005) only. For example (Chartrand, Kubicki, & Schultz 1998; Messmer & Bunke 1998) are based on graph and subgraph isomorphism. In (Du *et al.* 2005), an alphabet-based approach is proposed, basically restricting two workflows to a shared alphabet and revising them with the transformation rues introduced in (van der Aalst & Basten 2002). Our approach is an amalgam of both language comparison and structural comparison, as we first extract the common alphabet and then compare the languages both workflow produces w.r.t. the shared alphabet. Furthermore our alphabet captures the data flow which is ignored by all of the previous metrics.

Conclusions

This paper presented the WIT algorithm for inferring workflows from action traces without the need for multiple examples or action semantics. The utility of WIT was validated empirically in a web services-based domain. Future work will focus on empirical evaluation in additional domains, theoretical characterization of the conditions under which WIT is guaranteed to converge on a minimal workflow and on the effects of noisy action sequences (containing, for example, missteps on the part of the expert) and ways of dealing with them.

Acknowledgements

This work is supported by the DARPA Integrated Learning Program through a sub-contract from BBN.

References

Agrawal, R.; Gunopulos, D.; and Leymann, F. 1998. Mining process models from workflow logs. In *6th Int. Conf. on Extending Database Tech. (EDBT '98)*, 469–483.

Bauer, M. A. 1979. Programming by examples. *Artificial Intelligence* 12(1):1–21.

Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The Semantic Web. *Scientific American* 284(5):34–43.

Chartrand, G.; Kubicki, G.; and Schultz, M. 1998. Graph similarity and distance in graphs. *Aequationes Mathematicae* 20(1):129–145.

Cook, J. E., and Wolf, A. L. 1998. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.* 7(3).

Du, Z.; Huai, J.; Liu, Y.; Hu, C.; and Lei, L. 2005. Ipr: Automated interaction process reconciliation. In *IEEE Int. Conf. on Web Intelligence (WI'05)*, 450–456.

Lau, T. A., and Weld, D. S. 1999. Programming by demonstration: An inductive learning formulation. In *Intelligent User Interfaces*, 145–152.

Mahleko, B.; Wombacher, A.; and Fankhauser, P. 2005. A grammar-based index for matching business processes. In *IEEE Int. Conf. on Web Services (ICWS 2005*, 21–30.

Messmer, B. T., and Bunke, H. 1998. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. on Patt. Analysis and Mach. Intell.* 20(5):493–504.

Mohri, M. 2003. Edit-distance of weighted automata: General definitions and algorithms. *Int. J. Found. Comput. Sci.* 14(6):957–982.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *ICML '06*, 665–672.

Oates, T.; Desai, D.; and Bhat, V. 2002. Learning k-reversible context-free grammars from positive structural examples. In *ICML*, 459–465.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for web service composition using SHOP2. *Journal of Web Semantics* 1(4):377–396.

van der Aalst, W. M. P., and Basten, T. 2002. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.* 270(1-2):125–203.

Weijters, T., and van der Aalst, W. 2001. Process mining: Discovering workflow models from event-based data. In *13th Belgium-Netherlands Conf. on AI (BNAIC'01)*.

Winner, E., and Veloso, M. M. 2003. Distill: Learning domain-specific planners by example. In *20th Int. Conf. on Machine Learning (ICML 2003)*, 800–807.