A Context Driven Approach for Workflow Mining

Fusun Yaman
BBN Technologies
10 Moulton Street
Cambridge MA 02138 USA
fusun@bbn.com

Tim Oates
Department of Computer Science and
Electrical Engineering, UMBC
Baltimore MD 21250 USA
oates@cs.umbc.edu

Mark Burstein
BBN Technologies
10 Moulton Street
Cambridge MA 02138 USA
burstein@bbn.com

Abstract

Existing work on workflow mining ignores the dataflow aspect of the problem. This is not acceptable for service-oriented applications that use Web services with typed inputs and outputs. We propose a novel algorithm WIT (Workflow Inference from Traces) which identifies the context similarities of the observed actions based on the dataflow and uses model merging techniques to generalize the control flow and the dataflow simultaneously. We identify the class of workflows that WIT can learn correctly. We implemented WIT and tested it on a real world medical scheduling domain where WIT was able to find a good approximation of the target workflow.

1 Introduction

Automation of routine procedures removes the necessity of human supervision and leaves more time for people to coordinate complex tasks that require human level intelligence which the AI researchers aspire and are yet to achieve. Over the last decade, workflows have been commonly utilized to automate business processes, grid computing applications and many more. There has been a great deal of research in designing workflows that can capture the control flow of composite and complex processes.

Unfortunately designing workflows is a burden for users. It is much easier for humans to demonstrate the solution than to state the solution declaratively. An answer for this knowledge acquisition bottleneck is called *workflow mining* or *process mining* which aims to learn the underlying workflow given some example traces, i.e steps followed in the workflow for a specific case.

In this paper, we present a novel workflow mining approach where we combine the control flow and dataflow reasoning to discover the workflow. Interestingly, in all of the previous work on workflow mining the dataflow aspect of the problem was casually ignored. This is not acceptable in many applications which are built on top of Web services that have well defined I/O behaviors. It is worth to note that other research communities closely related to workflow mining, such as programming by demonstration and web service composition, have studied the dataflow aspect of the problem. However the nature of these work are quite different than ours.

Our approach analyzes the data dependencies in the trace to discover the context of the actions that appear in the trace. Using the context information we can decide whether the two occurrences of the same action correspond to the same node in the workflow or not. As a result, unlike the previous work [van der Aalst *et al.*, 2004; Cook and Wolf, 1998b; Agrawal *et al.*, 1998], we are able to learn workflows with non-unique action nodes. Furthermore, the context discovery can easily be generalized to work with causal dependencies instead of data dependencies. Thus, the ideas presented in this work can be applied to other areas such as learning domain specific knowledge for AI planning problems.

In this paper, we present the algorithm *Workflow Inference From Traces (WIT)*. WIT identifies the context similarities of the trace elements based on the dataflow and uses model merging (a technique borrowed from grammar inference literature) to generalize the control flow and dataflow simultaneously. In addition WIT can decompose the learned workflow into a hierarchy of workflows representing self-contained complex processes each of which can be reused in future workflow designs.

We implemented WIT and deployed it as a component in POIROT [Burstein *et al.*, 2007], an integrated learning application initiated by DARPA, where several learners interact to solve different parts of a complex learning problem. The performance of the system has been evaluated using a real world medical scheduling domain. Despite the additional complexity of the domain, WIT was able to learn the correct control flow as well as the dataflow in most of the cases.

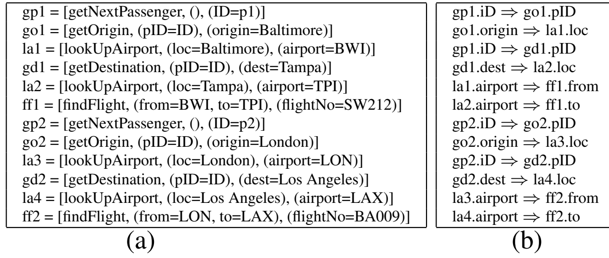
In addition to the empirical results, we present theoretical results on learnability of the workflows using WIT. Specifically, we show that the class of workflows that WIT can learn is a subset of reversible regular grammars [Angluin, 1982].

2 Definitions

The building blocks of workflows are actions. An action is either *simple*, i.e. executed as one step, or *composite*, i.e. accomplished in more than one ways or steps. The actions we consider have typed parameters. For any action A, the inputs and outputs of the action is represented as $In(A)$ and $Out(A)$ respectively. We will use $IO(A)$ to denote $In(A) \cup Out(A)$. We assume the parameter types have a semantic component (i.e. concepts in an ontology).

An instantiation of an action A maps the elements of $IO(A)$ to values consistent with the type of each I/O element. We use the notation $a = [A, (in_1 = v_1, \ldots, in_n = v_n), (out_1 = w_1, \ldots, out_m = w_m)]$ to denote that a is an instance of the action A where each input in_i is mapped to v_i and output out_i is mapped to w_i. For example $[lookUpAirport, (loc = Baltimore), (airport = BWI)$ is an instance of $lookUpAirport$ action that returns BWI as the closest airport to Baltimore.

A *trace* is a sequence of action instances, such as the one in Figure 1(a) which shows the steps of finding flights for two passengers. The data dependencies in a trace are the set of equality constraints in the form of $a.o \Rightarrow b.i$ which states the output o of the action instance a supplies the value for the input i of the instance b. Figure 1(b) lists all the data dependencies in the trace in Figure 1(a).


gp1 = [getNextPassenger, (), (ID=p1)]
go1 = [getOrigin, (pID=ID), (origin=Baltimore)]
la1 = [lookUpAirport, (loc=Baltimore), (airport=BWI)]
gd1 = [getDestination, (pID=ID), (dest=Tampa)]
la2 = [lookUpAirport, (loc=Tampa), (airport=TPI)]
ff1 = [findFlight, (from=BWI, to=TPI), (flightNo=SW212)]
gp2 = [getNextPassenger, (), (ID=p2)]
go2 = [getOrigin, (pID=ID), (origin=London)]
la3 = [lookUpAirport, (loc=London), (airport=LON)]
gd2 = [getDestination, (pID=ID), (dest=Los Angeles)]
la4 = [lookUpAirport, (loc=Los Angeles), (airport=LAX)]
ff2 = [findFlight, (from=LON, to=LAX), (flightNo=BA009)]
(a)

gp1.iD ⇒ go1.pID
go1.origin ⇒ la1.loc
gp1.iD ⇒ gd1.pID
gd1.dest ⇒ la2.loc
la1.airport ⇒ ff1.from
la2.airport ⇒ ff1.to
gp2.iD ⇒ go2.pID
go2.origin ⇒ la3.loc
gp2.iD ⇒ gd2.pID
gd2.dest ⇒ la4.loc
la3.airport ⇒ ff2.from
la4.airport ⇒ ff2.to
(b)

Figure 1: (a)Trace (b) Data dependencies in (a)

A *control flow* is a directed graph with two special nodes designated as the *start* and the *end* nodes. In a control flow every node, other than the start and end nodes, has an action associated with it. We will use $Action(n)$ to denote the action associated with a node n. An arc (n_1, n_2) signals that n_1 has to be executed before n_2. If a node has more than one successors then during the execution of the control flow only one of the edges will be followed. In this work we restrict ourselves to sequentially executed workflows. Thus we don't allow parallel execution of multiple branches. The top graph in Figure 2 is of a control flow that can produce the trace in Figure 1.

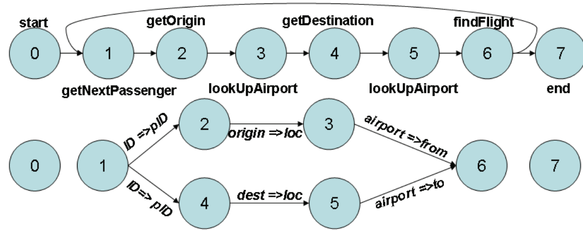


Figure 2: A Control flow(top) and its dataflow (bottom).

Note that the control flow in Figure 2 gives us only the accepted sequence of actions but it does not tell us how the I/O of each node is wired. A *dataflow* associated with a control flow C, is another directed graph in which there is a node for every node in the control flow. For any dataflow node n, $cNode(n)$ denotes the node in C that n is mapped to. Similarly, for a node n in C, $dNode(n)$ denotes the node in D that n is mapped to. Since there is a one to one mapping from dataflow nodes to control flow nodes we can overload the definition of $Action(n)$ for dataflow nodes, to act as a shortcut notation to $Action(cNode(n))$. Every arc (n_1, n_2) in the dataflow has a label of the form $o \Rightarrow i$ meaning the output o of $Action(n_1)$ produces the value for the input i of $Action(n_2)$. Any arc originating from the start node represents an input of the workflow, similarly any arc entering the end node is an output of the process. The bottom graph in Figure 2 is a dataflow for the control flow above. The mapping of the nodes is evident by the node numbers. It is easy to verify that the data dependencies Figure 1(b)are consistent with the arc labels in this dataflow. Note that the dataflow does not have to be connected.

A workflow is a tuple $W = \langle C, D\rangle$ where C is a control flow and D is a dataflow for C. An execution is a tuple $E = \langle T, B\rangle$ where T is a trace and B is a set of data dependencies in T. Furthermore W *produces* the execution E (equivalently E is a *production* of W) if C can generate the action instance sequence in T and D can generate data dependencies in B.

Definition 1 *A workflow* $W = \langle C, D\rangle$ *produces an execution* $E = \langle [a_1, \ldots, a_k], B\rangle$ *iff* C *has a path* $[start(W), n_1, \ldots, n_k, end(W)]$ *such that*

- *every* a_i *is an instantiation of* $Action(n_i)$ *and*
- *for every constraint* $a_i.o \Rightarrow a_j.i$ *in* B *there is an edge* (n_i, n_j) *in* D *with label* $o \Rightarrow i$*.*

where $start(W)$ */*$end(W)$ *is the start/end node in* C*.*

Finally the workflow mining problem is to discover a workflow that can produce a given a set of executions.

3 WIT Algorithm

Our solution to the workflow mining problem as described in the previous section, is the **WIT** (**W**orkflow **I**nference from **T**races) algorithm which generalizes the traces and data dependencies simultaneously by applying techniques from grammar inference literature.

The WIT algorithm has 3 steps; *initialization, context inference* and *step generalization.* The input to WIT is a set of executions (i.e. pairs of trace and data dependency sets). The initialization step transforms the input into a very specific workflow that can only produce the input executions. The context inference step analyzes the similarities between the data dependencies of the dataflow nodes. Step generalization, merges some of the states in the control flow and the dataflow based on their context and proximity. Step generalization can discover additional context similarities. If so WIT goes back to context inference step. WIT continues this loop until no more new context inferences can be made and outputs the workflow it has generalized so far. The following subsections explain the context inference and step generalization steps in detail.

3.1 Context Inference

The goal of context inference is to determine which nodes in a workflow are similar to each other. Nodes are similar if they represent same type of actions and their inputs/outputs

are supplied/used by similar actions. In other words, nodes with same kind of data dependencies are similar. We will use the concept of *context* to capture this similarity notion. More formally, given a dataflow $D = (N, E)$ with nodes N and edges E, *context* is a mapping of each node $n \in N$ and edge $e \in E$ to a token, denoted $\mathcal{C}(x) = \alpha$. A context for a dataflow $\mathcal{D}$ is *closed* if it satisfies all of the following four context axioms:

- $\forall e_1, e_2 \in E \; prod(e_1) = prod(e_2) \Longrightarrow \mathcal{C}(e_1) = \mathcal{C}(e_2)$
- $\forall e_1, e_2 \in E \; cons(e_1) = cons(e_2) \Longrightarrow \mathcal{C}(e_1) = \mathcal{C}(e_2)$
- Let n_1, n_2 be two nodes in N such that $Action(n_1) = Action(n_2) = A$. Then $\mathcal{C}(n_1) = \mathcal{C}(n_2)$ iff for every input $i \in In(A)$ there are edges e_1, e_2 such that $\mathcal{C}(e_1) = \mathcal{C}(e_2)$ and $cons(e_1) = (n_1, i)$ and $cons(e_2) = (n_2, i)$.
- Let n_1, n_2 be two nodes in N such that $Action(n_1) = Action(n_2) = A$. Then $\mathcal{C}(n_1) = \mathcal{C}(n_2)$ iff for every output o of A there are edges e_1, e_2 such that $\mathcal{C}(e_1) = \mathcal{C}(e_2)$ and $prod(e_1) = (n_1, o)$ and $prod(e_2) = (n_2, o)$.

where for any edge $e = (s_1, s_2)$ with label $o \Rightarrow i$, $prod(e) = (s_1, o)$ and $cons(e) = (s_2, i)$. Intuitively $prod(e)$ and $cons(e)$ are the producer and the consumer of a value. Since there can be more than one action with the same input (i.e. name and type), $cons$ has to identify the node the input belongs to. For the same reason $prod$ is a pair of a node and an output.

Basically the first condition states that all edges providing a value to an input must have the same context. The second condition is the counterpart for outputs. The third condition enforces that the nodes will have the same context token if and only if all of their inputs are provided by same context edges. The last condition ensures that the nodes with same context will have the same context assignment on their outgoing edges per output.

Given a dataflow D, the *base context* of D is a closed context with maximum number of unique mappings. WIT computes a base context, by assigning a unique token to every node and edge and then iteratively merging the tokens (i.e. replace all occurrences of one with the other) that violate the context axioms. Afterwards, WIT refines the base context by postulating additional equality constraints. These equality constraints are discovered during step generalization. Overall, WIT refines a context C for a given dataflow D and context constraints X using Algorithm 1 as shown.

Algorithm 1 $Refine(\mathcal{D}, \mathcal{C}, \mathcal{X})$

Inputs: Dataflow $\mathcal{D}$, context $\mathcal{C}$, set of equality constraints $\mathcal{X}$.
```
while X is not empty do
    Remove a constraint t1 = t2 from X
    Replace every occurrence of t1 in C and X by t2.
    while C violates context axioms do
        Find tokens α and β violating context axioms w.r.t D
        Replace every occurrence of α in C and X by β.
return C
```

Example 1 *Let T be a trace with the following instance sequence:* $[A, (), (o = 1)], [B, (i = 1), (o = 2)], [D, (i = 2), ()], [A, (), (o = 3)], [C, (i = 3), (o = 4)], [D, (i = 4), ()], [A, (), (o = 5)], [C, (i = 5), (o = 6)], [D, (i = 6), ()]$. *Figure 3(i) shows the dataflow for a control flow which chains nine action nodes to produce T. Also the actions associated with each node is shown in the nodes. The base context is shown in (ii)where the context tokens are shown in the nodes and under the edges. Finally when we revise the base context with the constraint $d1 = d2$, we get the context in (iii).*

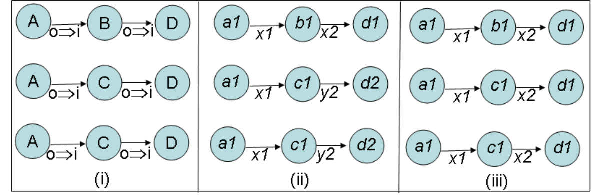


Figure 3: Dataflow, base context and revised contexts.

3.2 Step generalization

This step generalizes a workflow W by merging some of the states in its control flow and dataflow. Since there is a one-to-one correspondence between the nodes in a control flow and a dataflow, when two states are merged in one graph, the corresponding states in the other graph is also merged. When nodes n_1 and n_2 are merged, all edges incoming/outgoing to/from n_2 are redirected to n_1 and n_2 is removed. There are two kinds of merge:

- *Step merging*, is applicable if two dataflow states d_1 and d_2 have the same context. Step merging simply merges the nodes d_1 and d_2 and the corresponding states in the control flow and no new context constraint is discovered.
- *Context merging* step, is applicable if two control flow states w_1 and w_2 have a common predecessor or a common successor and $Action(w_1) = Action(w_2)$. Let d_1 and d_2 be the dataflow states for w_1 and w_2. Context merging requires the context to satisfy the equality constraint $\mathcal{C}(d_1) = \mathcal{C}(d_2)$.

Step merges are useful for identifying the loops in a control flow even when there is only one trace to learn from. Context merges on the other hand deemphasizes the context difference between two action instances if they are close enough in the control flow. Deemphasizing prevents the control flow from being overly specific.

Algorithm $Generalize(\langle W, D\rangle, C)$, starts with an empty set of constraints X and performs all the step merges on dataflow D based on the context $\mathcal{C}$. Then context merges on control flow W are performed as the constraints X are updated. It returns the generalized workflow along with a possibly empty set of constraints X.

Figure 4 demonstrates the control flows (left) and the dataflows (right) obtained after the step merges (top) and context merges (bottom) for the trace in Example 1. In the dataflows, instead of the arc labels, which are all $o \Rightarrow i$, the context for each node and edge are shown. Note that the context tokens $x2$ and $y2$ violate the first context axiom. They

Algorithm 2 $Generalize(\langle\mathcal{W},\mathcal{D}\rangle,\mathcal{C})$

Inputs: A workflow with control flow $\mathcal{W}$ and dataflow $\mathcal{D}$, context $\mathcal{C}$.
 while step merge is applicable **do**
  Pick any two nodes d_1,$d_2 \in \mathcal{D}$ such that $\mathcal{C}(d_1) = \mathcal{C}(d_2)$
  Merge nodes d_1 and d_2 in $\mathcal{D}$.
  Merge nodes $cNode(d_1)$ and $cNode(d_2)$ in $\mathcal{W}$.
 $\mathcal{X} = \{\}$
 while context merge is applicable **do**
  Pick two nodes w_1,$w_2 \in \mathcal{W}$ with a common predecessor or successor and $Action(w_1) = Action(w_2)$.
  Merge nodes w_1 and w_2 in $\mathcal{W}$.
  Add constraint $\mathcal{C}(w_1) = \mathcal{C}(w_2)$ to $\mathcal{X}$.
  Merge nodes $dNode(w_1)$ and $dNode(w_2)$ in $\mathcal{D}$.
 return $\langle\mathcal{W},\mathcal{D}\rangle$,$\mathcal{X}$

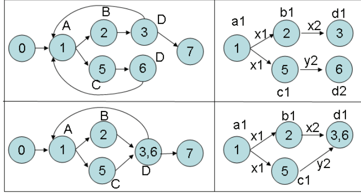


Figure 4: Control flows (left), dataflows and context (right) for the trace in Example 1, after step merging (top) and context merging (bottom).

will be merged into a single token when the algorithm goes into context inference step again.

If new context constraints are discovered during the merges, then WIT goes back to the context inference step with an updated constraint list. Otherwise WIT outputs the workflow after all the merges are completed.

4 Hierarchical Decomposition

We can also simplify a workflow by identifying subgraphs (representing branches or loops) in the control flow and replacing them with single a node that is associated with a new composite action. Every composite action has a workflow, denoted $wf(A)$ which details the internals of the action. By applying the same decomposition algorithm to the workflows of discovered actions we end up with a hierarchy of composite actions.

The procedure $Decompose(\langle\mathcal{W},\mathcal{D}\rangle)$ (shown in Algorithm 3) takes the workflow $\langle\mathcal{W},\mathcal{D}\rangle$ as input. It first identifies and processes the loops and then moves on to branches. The loop identification assumes that for every loop there are two nodes n_1 and n_2 in $\mathcal{W}$ such that every iteration of the loop starts with n_1 and ends with node n_2. For any two nodes n_1 and n_2 such that all paths from n_1 visits n_2 and there is an edge n_2, n_1 there is a loop in the control flow. Furthermore n_2 is a sink of n_1, denoted $sink(n_1)$ and n_2 is a source for n_2, denoted $source(n_1)$. So loop identification requires two control flow nodes n_1, n_2 such that $sink(n_1) = n_2$ and an edge (n_2, n_1) creating the loop. Then we replace all nodes that appear in a path from n_1 to n_2, using $Replace$ function as shown in Algorithm 4 and apply $Decompose$ procedure to the newly extracted workflow. Identifying branches is similar, only this time we require the source node to have more than one outgoing edges and we do not include the source and sink nodes in the set that is to be replaced.

Algorithm 3 $Decompose(\langle\mathcal{W},\mathcal{D}\rangle)$

Inputs: Control flow $\mathcal{W} = (N_W, E_W)$ and dataflow $\mathcal{D}$
 for every $n_1, n_2 \in N_W$ such that $sink(n_1) = n_2$ and $e = (n_2, n_1) \in E_W$ **do**
  $S = \{n \in N_W \mid \text{appear in a path from } n_2 \text{ to } n_1\}$
  $(\langle\mathcal{W},\mathcal{D}\rangle, A) = Replace(\langle\mathcal{W},\mathcal{D}\rangle, S, e)$
  $(\langle w, d\rangle, actionList) = Decompose(wf(A))$
  $newActions = newActions \cup actionSet \cup A$
 for each $n_1, n_2 \in N_W$ such that $sink(n_1) = n_2$ and $outDegree(n_1) > 1$ **do**
  $S = \{n \in N_W \mid \text{appear in a path from } n_1 \text{ to } n_2\}$
  $S = S - \{n_1, n_2\}$
  $(\langle\mathcal{W},\mathcal{D}\rangle, A) = Replace(\langle\mathcal{W},\mathcal{D}\rangle, S, nil)$
  $(\langle w, d\rangle, actionList) = Decompose(wf(A))$
  $newActions = newActions \cup actionSet \cup A$
 return $\mathcal{W}$,$\mathcal{D}$ and $newActions$.

The procedure $Replace(\langle\mathcal{W},\mathcal{D}\rangle, S, loopEdge)$ replaces the nodes S from the control flow $\mathcal{W}$ and the dataflow nodes corresponding to nodes in S, denoted D_S, from $\mathcal{D}$ with new nodes n_w and n_d respectively. The procedure redirects the edges in $\mathcal{W}$ and $\mathcal{D}$ to the new nodes. A new composite action A is created such that for every incoming/outgoing edge e to/from a node in D_S from/to outside of D_S there is an input/output in $In(A)/Out(A)$. Furthermore let e_1 and e_2 be such incoming/outgoing edges, if $\mathcal{C}$ is the base context for $\mathcal{D}$ and $\mathcal{C}(e_1) = \mathcal{C}(e_2)$ then the edges are mapped to same input/ output of A. This condition ensures that the action A won't have any duplicate or unnecessary I/O. The $Replace$ procedure assumes that composite actions representing loops do not have any outputs.

5 Theoretical Results

In this section we define a class of workflows which WIT can learn correctly and completely. We use $WIT(S)$ to denote the workflow that was learned by WIT given S, a set of productions (instances) of a target workflow.

Definition 2 (Witty workflow) *A workflow $\langle W, D\rangle$ is witty iff it satisfies the following conditions:*

- *Let $\mathcal{C}$ be the base context for D. For any two nodes d_1, d_2 in D, $\mathcal{C}(d_1) = \mathcal{C}(d_2)$ implies $d_1 = d_2$, i.e. the dataflow has unique nodes per context token.*
- *If any two nodes w_1 and w_2 in W has a common predecessor or successor then $Action(w_1) \neq Action(w_2)$.*

Next we are going to explore the relationship between witty workflows and reversible regular grammars [Angluin, 1982]. The importance of this relationship lies in the fact that, grammar induction is a special case of workflow mining (i.e., when actions have no parameters). It has been shown that in the absence of negative-examples the target grammar

Algorithm 4 $Replace(\langle \mathcal{W}, \mathcal{D} \rangle, S, loopEdge)$

Inputs: Workflow with control flow $\mathcal{W} = (N_W, E_W)$ and dataflow $\mathcal{D} = (N_D, E_D)$, $S \subset N_W$ and $loopEdge \in E_W$
 S_D contains the dataflow nodes for nodes in S
 Add new nodes n_w and n_d to $\mathcal{W}$ and $\mathcal{D}$.
 Let $\mathcal{C}$ be the base context for $\mathcal{D}$.
 $A = CreateNewAction(\mathcal{D}, D_S, \mathcal{C})$
 $Action(n_w) = A$ and $cNode(n_d) = n_w$
 if $loopEdge \neq nil$ **then**
  Remove $loopEdge$ from E_W and add edge (n_w, n_w)
 $w = subControlflow(W, S)$
 Remove every edge e from E_W, if both ends of e are in S
 for every $e = (n_1, n_2) \in E_W$ such that $n_2 \in S$ **do**
  Add edge (n_1, n_w) to E_W
 for every $e = (n_1, n_2) \in E_W$ such that $n_1 \in S$ **do**
  Add edge (n_w, n_2) to E_W
 $d = subDataflow(D, D_S)$
 Remove every edge e from E_D, if both ends of e are in D_S
 for every $e = (n_1, n_2) \in E_D$ such that $n_2 \in D_S$ **do**
  Let $o \Rightarrow i$ be the label of e
  Let $i' \in (A)$ such that i' is associated with $\mathcal{C}(e)$
  Add edge $e' = (n_1, n_d)$ to E_D with label $o \Rightarrow i'$
 for every $e = (n_1, n_2) \in E_W$ such that $n_1 \in D_S$ **do**
  Let $o \Rightarrow i$ be the label of e
  Let $o' \in (A)$ such that o' is associated with $\mathcal{C}(e)$
  Add edge $e' = (n_d, n_2)$ to E_D with label $o' \Rightarrow i$
 $wf(A) = \langle w, d \rangle$
 return $\langle \mathcal{W}, \mathcal{D} \rangle$, and A

can be learned if it is reversible [Angluin, 1982]. In workflow mining we get only positive examples so we are constrained by this result as well. Theorem2 states that witty workflows are a subset of reversible grammars. A regular grammar G is reversible iff the productions of G satisfy the following two conditions: (i)If $A \rightarrow aB$ and $A \rightarrow aC$ then $B = C$ and (ii)$A \rightarrow aC$ and $B \rightarrow aC$ then $A = B$, where A, B and C are nonterminals and a is a terminal.

Theorem 1 *Let $\langle W, D \rangle$ be a witty workflow then there is a reversible regular grammar $G = (T, N, P)$ with terminals T, nonterminals N and production rules P such that (i) For every node in W there is a non terminal in N (ii) For every action in W, there is a terminal in T(iii) $A \rightarrow aB$ is in P iff there is an arc (s, w) in W such that A, B and a are the corresponding non-terminal and terminal symbols for s, w and $Action(s)$.*

The correctness of the theorem is entailed by the second condition in Definition 2.

Theorem 2 (soundness/completeness) *Let W be a witty workflow. Then*

- *WIT is sound, i.e. for any set S of productions of W, every production of $WIT(S)$ is a production of W.*
- *WIT is complete, i.e. there is a finite set S of of productions of W such that all productions of W are productions of $WIT(S)$.*

Correctness of WIT can be proven by induction where base case is the workflow representing the input traces which are by definition productions of W. The inductive step shows that with every merge the previous workflow converges to (W, D). The general idea of the completeness proof is to show that for any witty workflow W there is an S containing a finite number of productions that traverses all edges of the controlflow and the dataflow.

6 Implementation and Empirical Results

We implemented WIT in Java. The implementation has some extra features. The most notable of all being the data dependency extraction which analyses the inputs and outputs of trace elements and extracts the data dependency information using a set of domain independent heuristics. In many domains traces are available (e.g., in the form of execution logs) however the data dependencies are not explicitly provided, i.e. need to be inferred from the I/O of the trace elements. Data dependency extraction feature allows WIT to run in domains where data dependencies are not provided.

We have deployed WIT as a component in POIROT, a framework built to take DARPA's integrated learning challenge, where several learners interact to solve different parts of a complex learning problem. In this framework WIT is the learner responsible for discovering the workflow given a single execution. Other components in POIROT are responsible for learning expert preferences, preconditions/effects of composite actions and finally conditions for branches and loops in the control flow.

POIROT is tested using a real world medical transportation/scheduling domain where the wounded soldiers are transported outside of the combat zones into safe areas. The system is presented an expert trace that demonstrates the transportation of 4 patients. The expert trace contains 126 actions and 337 data bindings. The data dependency information is not explicitly provided so WIT uses the data dependency extraction step. After the learning phase, POIROT solves 20 new problems (each containing 4 patients)and outputs a trace per problem. Output traces are aligned with the expert solution traces and the percentage of correct action steps and correct parameter assignments are computed (in addition to other metrics). Note that these two metrics demonstrate the performance of WIT as well. In these experiments the average action step accuracy of POIROT was 92 percent and the parameter assignment accuracy was 82 percent. For novice humans that were subject to the same test, the numbers were 70 and 76 percent respectively where the difference was statistically significant.

7 Related Work

There is a large body of work in workflow/process mining. Most of these work [van der Aalst *et al.*, 2004; Cook and Wolf, 1998a; Agrawal *et al.*, 1998], can discover non-sequential workflows but they can not handle workflows containing more than one node per action. Among the existing work some [Cook and Wolf, 1995; Herbst and Karagiannis, 1998] employ grammar inferencing techniques to find the target workflow. Cook and Wolf [1995], groups the actions that have the same k-step future into the same node.

The major problem with this approach is figuring out the parameter k. Usually a correct guess requires prior knowledge about the structure of the target workflow. Herbst and Kragiannis [1998] uses Bayesian model merging with the aim of maximizing the likelihood of the model. Unlike WIT, this aproach requires more than one example to learn loops.

WIT can also decompose workflows using a simple structure-driven approach which is significantly different from the taxonomy-based approach described in [Greco *et al.*, 2005]. Our approach does not involve search for detecting the best possible decomposition in terms of grouping the actions that are used for achieving a common goal. Although WIT's decomposition is computationally in expensive, it is applicable to certain kinds of graphs.

Programming by demonstration is a research area that is closely related to our work. Shen et al. [2009] describes a domain specific system that identifies frequent patterns in a dataflow to identify actions related to the same task. Extracted actions are then fed into an ordinary workflow learning algorithm to learn macros actions. Unlike our approach this work is useful for generalizing parts of the observations. Furthermore identifying the dataflow patterns boils down to graph isomorphism problem which is different and harder than inferring context similarities.

In general provenance is utilized in web services research to track the quality of service however some work goes beyond that. Leake and Kendall-Morwick [2008] propose a system to facilitate workflow generation. Even though the system does not discover workflows, it uses the dataflow to determine the similarities between workflows. The similarity evaluation is based on matching dataflow patterns which again is a form of graph isomorphism. Another work [Dustdar and Gombotz, 2007] uses detailed logs to learn workflows composed of webservices. The log includes certain types of interactions between services and is used for identifying subsequences of a trace that correspond to executions of different composite services.

8 Conclusions

In this paper, we presented a novel workflow mining algorithm WIT, which combines the control flow and dataflow reasoning to approximate the target workflow. WIT uses model merging to generalize the control flow and dataflow simultaneously. In addition, we have presented a hierarchical decomposition algorithm which simplifies a given workflow by recursively identifying and replacing complex subgraphs with composite processes, allowing them to be reused in future workflow designs. We have identified a class of workflows for which WIT is sound and complete. Our implementation is tested with target workflows that were outside of the class we have identified, WIT was able to approximate most of the characteristics of the target workflows. Future work will generalize WIT for parallel workflows by detecting the controlflow elements that don't have a data dependency. Also we think that decomposition can help us discover additional context similarities and can lead to further generalization of the workflow.

Acknowledgments

This project is supported by DARPA IPTO under contract FA8650-06-C-7606. Approved for Public Release, Distribution Unlimited.

References

[Agrawal *et al.*, 1998] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. *Lecture Notes in Computer Science*, 1377, 1998.

[Angluin, 1982] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, 1982.

[Burstein *et al.*, 2007] M. Burstein, M. Brinn, M. Cox, T. Hussain, R. Laddaga, D. McDermott, D. McDonald, and R. Tomlinson. An architecture and language for the integrated learning of demonstrations. In *AAAI Workshop Acquiring Planning Knowledge via Demonstration*, pages 6–11, 2007.

[Cook and Wolf, 1995] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *Proc. of ICSE '95*, pages 73–82, New York, NY, USA, 1995.

[Cook and Wolf, 1998a] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *Proc. of SIGSOFT '98/FSE-6*, pages 35–45, New York, NY, USA, 1998. ACM.

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

[Dustdar and Gombotz, 2007] Schahram Dustdar and Robert Gombotz. Discovering web service workflows using web services interaction mining. *International Journal of Business Process Integration and Management*, 1:256–266(11), 2007.

[Greco *et al.*, 2005] Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. Mining hierarchies of models: From abstract views to concrete specifications. In *Business Process Management*, pages 32–47, 2005.

[Herbst and Karagiannis, 1998] J. Herbst and D. Karagiannis. Integrating machine learning and workflow management to support acquisition and adaptation of workflow models. In *Proc. of DEXA 98*, Washington, DC, USA, 1998. IEEE Computer Society.

[Leake and Kendall-Morwick, 2008] David B. Leake and Joseph Kendall-Morwick. Towards case-based support for e-science workflow generation by mining provenance. In *ECCBR*, volume 5239 of *Lecture Notes in Computer Science*, pages 269–283, 2008.

[Shen *et al.*, 2009] Jianqiang Shen, Erin Fitzhenry, and Thomas G. Dietterich. Discovering frequent work procedures from resource connections. In *IUI*, pages 277–286. ACM, 2009.

[van der Aalst *et al.*, 2004] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, 2004.