ABSTRACT
The POIROT project is a four-year effort to develop an architecture that integrates the products of a number of targeted reasoning and learning components to produce executable representations of demonstrated web service workflow processes. To do this it combines contributions from multiple trace analysis (interpretation) and learning methods guided by a meta-control regime that reviews explicit learning hypotheses and posts new learning goals and internal learning subtasks. POIROT’s meta-controller guides the activity of its components through largely distinct phases of processing from trace interpretation, to inductive learning, hypotheses combination and experimental evaluation. In this paper we discuss the impact that various kinds of inference during the trace interpretation phase can have on the quality of the learned models.

Categories and Subject Descriptors
Learning, knowledge acquisition, analogies, planning, procedures.

General Terms
Algorithms, Performance, Design

Keywords
Multi-strategy learning, meta-control, integration.

INTRODUCTION
POIROT (Plan Order Induction by Reasoning from One Trial) is an architecture and strategy for applying a suite of learning and inference components learn web service procedures from one or a small set of examples. The system acquires and test executes its declarative representations of the learned procedures using semantic web service techniques based on OWL-S[15]. The system’s meta-controller manages a set of active learning goals representing from its component’s information needs and gaps in the learned models revealed by an evaluation of the combined intermediate products they individually produce.

A trace for POIROT is a semantic description of a sequence of web service calls represented in terms of semantic definitions of the services. Each trace element includes the service action’s type, its input and output parameters and a predicate calculus representation of the action’s effects such as is commonly found in an AI planning system. POIROT’s meta-control system guides the learning and inference components through multiple phases of processing in order to learn procedures. In the first, the trace interpretation phase, components perform causal and data dependency analysis to find necessary dependencies between steps. Data dependencies relate service outputs to subsequent service inputs. Causal dependencies are based on the use of background knowledge, and allow POIROT to postulate unobserved steps such as choosing a preferred element from a set of outputs. Another component clusters related steps that address likely subgoals. Overall, the interpretation phase embodies and extends a set of heuristics and inferences that were scattered in learning components during Phase I of the project, but are important in preparing for workflow generalization.

Subsequent phases of processing include workflow generalization, preference learning, model combination, experimentation and testing. POIROT tests its own conclusions on new problems by invoking the learned methods which results in calls to the same web services it observed the demonstrator using. Many of these elements were described in [4]. This paper emphasizes the role that trace interpretation has played in improving POIROT’s learning performance.

Learning components that operate on the interpreted trace to form procedures include several kinds of step generalization, loop identification, iteration order learning, preference and branch condition learning and method extraction. Collectively, the learning components create multiple hypotheses about the methods being learned, feeding POIROT’s method synthesis (stitching) process.

After Phase I of the project [4] POIROT was achieving performance on the learning task that was roughly 90% of
novice human performance in learning to perform a task where both the human subjects and POIROT saw one demonstration of how to schedule four patients for transport, a process requiring 30 service calls. After Phase II POIROT achieved slightly better than novice human performance while learning a more difficult version of the learning task involving 111 service calls for four patients.

This paper gives an overview of the Phase II system and describes some of the major improvements. In particular, a significant part of POIROT’s ability to learn more complex procedures is based on improvement and restructuring of the process of interpreting traces to reveal their organization and dependency structure before forming generalized procedures. Previously, only some of these techniques were used, and then only by particular learning components.

OVERVIEW OF LEARNING TASK

POIROT’s objective is to form a generalized hierarchical task model from ‘observed’ semantic traces of sequences of web service transactions. Our test domain is medical evacuation airlift planning, which is analogous to using planning a trip using different web sites to book airline tickets, reserve local transportation (e.g., taxis, trains buses) and secure hotel rooms. The domains are different in that planning medical evacuations can include requesting that new flights be established if an existing one won’t do, and making choices among helicopters and ambulances rather than trains and taxis. Medevac planning also includes making reservations for medical personnel and equipment to accompany patients on their journeys. A typical demonstration used as the basis for POIROT’s learning shows a user building a schedule for a number of patients with specific individual requirements. Each patient is available at a specified time and location to be moved, with a given destination hospital, latest arrival time and identified special needs (for medical care and equipment en route).

POIROT’s ‘observations’ are of a logged sequence of calls to web service calls collected when a user demonstrates the procedure by invoking the services individually through a GUI. When the results of these calls appear on the screen, they can be used to call other services by dragging them to fields in the chosen service form. For example, traces show users looking up airports by geographic location, and then finding available flights to and from those airports, and reserving seats on the chosen flights. Figure 1 shows a screenshot of the GUI used to call these services when developing a demonstration. During our evaluation process, when comparing POIROT to human performance, a video demonstration of this interface was presented to our human subjects as training before we asked them to try the task themselves.

Figure 2 shows the procedure that was demonstrated in our most recent set of experiments. Dotted arrows from boxes representing subtasks point to subprocedures that were also part of the demonstration. The increased difficulty of the task this year came partly from introducing subtasks for scheduling local transportation legs and for handling the special needs of patients en route. These subtasks and their irregularities caused multiple difficulties for the learning algorithms, as loops bodies more became irregular and there were even conditional loops within loops. While the outlines of the procedure being learned remained the same across patients, the number of steps and specific details often varied dramatically, and this made finding loops and conditionals very difficult.

Figure 1: Demonstration and human testing UI

Figure 2: Demonstrated Procedure

An important difference between the way human subjects and POIROT saw the demonstration is that human subjects saw a video demonstration using the interface shown in Figure 1, with explicit use of drag and drop actions that showed the exact screen sources of information used as input to services calls, while POIROT ‘observes’ only the semantic descriptions of traces captured from the HTTP or SOAP interactions between a client (e.g., a web browser) and a web server. As a result, POIROT determines how
data values used in service calls are arrived at by finding prior calls that produced those exact data values. This heuristic approach means that some data flow relationships interpreted as ambiguous by POIROT because the same values (e.g., location IDs, which are associated with many objects in the domain) appear multiple times in the trace history. POIROT’s dataflow interpretation process. IODA, uses a suite of domain independent heuristics to overcome this, but, there are cases where some background domain knowledge would be required for perfect accuracy. While there were far too many drag-and-drop actions for people to remember them all, our subjects could try to make sense of specific dragging actions as they went along.

ARCHITECTURE AND PROCESSING

POIROT’s software architecture uses a combination of agent and semantic web service concepts to integrate multiple learning and execution components. Figure 3 shows the Phase II architecture. The shared knowledge blackboard is an RDF store (Sesame), utilizing a publish-subscribe interface to trigger POIROT components.

While the learning components of POIROT use different approaches to generate learning hypotheses, all hypotheses are ultimately represented using the Learnable Task Modeling Language (LTML)[16] which has a compact s-expression syntax but translates to OWL/RDF on the central blackboard. LTML is based in part on the semantic web service markup language OWL-S [3, 15]. Like OWL-S, LTML represents semantic web services and procedures composed from those services using semantic types for actions, service parameters and effect relations. It uses the OWL Virtual Machine (OVM) to call individual web services as seen in these demonstrations [3,15,22].

about the observed traces and the procedural models generalized from those traces in one language, POIROT can share them among components that internally represent things quite differently, and can integrate their products to form a result that is better than all of the individual ones by selectively combining consistent sub-hypotheses in a process we call “stitching”.

Figure 4 represents the general order of internal processing that POIROT now uses, as dependencies among the internal components used in the learning process.

Trace Interpretation Phase

The components involved in Trace Interpretation are:

- IODA – (Input-Output Data Analysis) does data dependency analysis to find the most likely associations between service outputs and subsequent service inputs.
- YAPPR – Uses intent inference techniques [13] to produce a ‘bracketing’ of the trace sequence into subsequences that are likely to be subtasks.
- XPLAIN – uses background knowledge in the form of explanation patterns [8] to identify subgoals and causal dependencies between steps and introduces ‘mental actions’ into the annotated trace to represent unobserved steps such as the act of choosing a single item to use from a set of outputs produced by a query service.

Data dependency analysis examines the inputs and the outputs of the observed actions to determine the likely producer of each action’s inputs. POIROT relies on data equality to identify sources for service inputs. It is important to note, though, that output data values may be buried in tables of data produced by service queries. The analysis
relies on domain independent heuristics to pick a ‘best’ producer when the trace contains multiple sources for the same value. For ambiguous cases, the analysis prefers data values that come from 1) the same source (service/output record) as others used by the service; 2) a similar source as an unambiguous input for the same parameter of another call to the service; or 3) the most recent producer.

The bracketing process in YAPPR uses background knowledge about the task, namely that the goal is to make travel reservations, to identify steps that achieve subgoals pertinent to those objectives (reservations for travel resources) and then groups steps together that do information gathering in service of those subgoals. It uses information from XPLAIN and IODA to identify the dependent steps.

XPLAIN’s introduction of mental choice operations motivates POIROT’s internal goal to learn preference models for those choice operations. It’s annotation of causal dependencies enables POIROT to reason about such things as reservations for transport.

**Generalization Phase**

The central learning components for process generalization are WIT and DISTILL, which take different approaches to forming procedures from example traces.

WIT [26, 28] produces finite state graph models of generalized procedures using techniques from grammar induction [20]. The models it produces preserve the order of steps observed in the traces. It finds all significant loops and branches, but does not determine the termination criteria on loops or the branch conditions at process forks.

DISTILL [24, 25], tries to group steps into loops eagerly, organized by known dependencies between steps. The procedures it produces are typically ordered differently than what was demonstrated, though they do reflect necessary ordering dependencies and include step pre-conditions that can be used as branch conditions during stitching [4].

CHARM [27] and DRILL [23] use different approaches to learn about preference decisions such as how to order iteration variable values for loops and how to select preferred objects from sets. For example, once WIT and DISTILL have discovered that there is a loop over the set of requirements to plan for each patient, DRILL is able to learn a loop ordering function based on information about the order in which the patients were processed in the demonstration trace. Specifically, it learns that triage code and nationality are used in sorting the patients for processing.

CHARM is learns object selection functions, such as how to pick an airport from a list of airports within some distance of a given location as returned by the service lookupAirport. Both CHARM and DRILL essentially learn ranking functions based on knowledge of ordinal properties associated with the objects being ranked, and a small number of examples of object ordering or selection extracted from a single demonstration trace.

**INTERPRETATION HELPS LEARNING**

We now discuss in more detail the impact of recent extensions of POIROT on its performance, focusing on how it takes advantage of trace interpretation results to improve its effectiveness for WIT in particular. WIT originally had its own variant of the IODA data dependency analysis, but now uses IODA results which incorporates many of WITs data dependency analysis heuristics. WIT also relies on XPLAIN to insert mental choice operations into the trace. It uses YAPPR’s trace bracketing to abstract away details of the trace so that it can learn the overall structure of the trace before forming specific submethods. We discuss the first and third of these in more detail below.

Data dependency analysis plays a key role in WIT’s ability to detect similar actions and recognize loops, allowing it to transform the trace into a generalized workflow. WIT produces better workflows on many traces if it uses bracketing information. It does this by replacing subsequences of a trace with abstract actions when it is first learning. This lets WIT learn the core of the target workflow with fewer examples. Similar arguments apply to DISTILL.

**WIT overview**

WIT learns a workflow given a set of observed traces. It uses grammar inferencing techniques (specifically model merging) to transform the input sequences into a more general graph containing branches and loops. The learned workflow can generate (in the sense that a grammar is generative) the demonstrated sequences as well as unseen sequences. The WIT algorithm has four steps; data dependency analysis (now shared with IODA), step similarity detection, process generalization and method decomposition. These steps are explained in [28] in detail.

Step similarity detection identifies similar actions in the trace by comparing the steps’ action types and the types of dependencies they have on other steps. This process discovers which of the actions in the trace should correspond to the same generalized node in the workflow. For example, two lookupAirport actions are judged similar if they both find airports near the origins of two patients, but different if one gets its input from the origin of a patient and the other from the destination of the patient.

Process generalization starts with a very specific workflow that can only generate the input trace and then iteratively merges nodes that are identified as similar in the similarity detection step. It also merges nodes (with the same action type) that satisfy certain proximity constraints (e.g., nodes that have the same immediate predecessor). The second type of merging is inspired by grammar inference algorithms which learn a class of regular grammars using only positive examples from the language. As it is possible to detect more similarities after a round of process generalization, the step similarity detection and process generali-
zation algorithms are interleaved until the set of similarities and the process graphs converge to a fixed point.

The final step, method decomposition, recursively identifies blocks of related steps in the learned graph and replaces them with single abstract nodes representing composite actions. At the end of this process we end up with a hierarchy of methods. The current decomposition algorithm can only decompose the workflow correctly if the workflow does not have any nested loops.

Figure 5a (top) shows a very simple version of the target POIROT workflow and a trace (5b, bottom) for two patients. The workflow finds the closest airports to each patient’s current and final locations, finds and reserves a flight between the airports and arranges local transportation to/from the airports. There are two types of local transportation, provided by the Army and the Marines. In the trace, the first patient is scheduled to be transported to his originating airport using an Army vehicle and is transported from his destination airport to final destination by the Marines. The second patient is scheduled to be transported by Marines first and then by the Army. Different services getArmyTransportTime and getMarineTransportTime (GAT/GMT) are called to find out (Army/Marine) transport times and to notify the chosen carrier (NotifyArmy, NotifyMarines = NA/NM).

Figure 5: A Simple workflow (5a) and a trace based on it with two patients (5b).

Figure 6 shows the workflow learned by WIT given the trace in Figure 5b, assuming the data dependency analysis step was correct and complete. The numbers on top of a node indicate which of the trace elements were merged into this node. Note that the learned workflow is very similar to the target workflow (Figure 5a) but missing some of the edges. WIT can learn all of the missing edges with additional example traces. Note that this workflow contains nested loops and therefore WIT’s decomposition step won’t produce the correct decomposition.

The effect of data dependency analysis on WIT output is critically important. Data dependency analysis is based on domain independent heuristics that decide between multiple candidate producers of information for a given consumer.

Collectively, the current heuristics are fallible because they do not use sufficient domain knowledge to reason about the true correspondences.

Improved learning using bracketing

If, instead of the original trace, we feed WIT a trace interpretation, where some subsequences of the original trace are replaced with abstract actions based on bracketing, we can improve the result in three ways:

1. Reducing the ambiguity in the data dependency analysis by masking the unnecessary outputs produced in subtasks.
2. Removing nested loops temporarily and replacing the nested loop iterations with an abstract action.

The output would have been closer to the best WIT workflow had the immediate predecessors of the 6th and the 18th actions been the same. In that case, WIT would conclude that the difference between those two elements (6,18) was negligible and would merge the two actions into the same node, ultimately removing the distinctions between the pairs of actions that followed (7,19 and 10,22), and leading to a result like Figure 6.

Figure 7: Learned workflow when WIT can not resolve the producer for an input of the fifth trace element.
3. Making the trace more regular by replacing different subsequences achieving the same goal with a single node.

Basically, (1) improves the data dependency analysis leading to better generalized workflows; (2) avoids the nested loop restriction of the decomposition algorithm, allowing the nested loops to be learned in a separate pass; and (3) allows WIT to learn the target workflow with fewer examples. We demonstrate these improvements with the help of another case study.

Figure 8 is an interpretation of the trace in Figure 5b. In this interpretation, subsequences [3,4,5], [6,7], [15,16,17], and [20,21] are replaced by abstract actions with the name lookupTransportTime, which is abbreviated as LTT. The actions in this bracket are used for finding the local transport unit that can reach the patient’s location fastest. Also note that the actions NM and NA which are used for notifying either a marine or an army unit have been replaced by a generic NAM action which is used for notifying any military unit.

Figure 9 demonstrates the workflow WIT learns from the interpreted trace and the subworkflows corresponding to the internal structure of the abstract actions LLT and NAM. WIT learns the workflows representing the internal structure of the abstract actions by simply using the bracketed subsequences as input traces. This process is managed by the POIROT meta-control system.

Now compare the two workflows in Figure 6 and Figure 9. Remember that the workflow in Figure 6 was pretty close to the target workflow but it was missing ten edges. If we inline the workflows of the abstract actions into the workflow in Figure 9 we will get the workflow in Figure 10 which now includes six of those missing edges. Note that even though the original trace did not contain evidence of a loop after the second LT action, because the interpreted trace was more regular, WIT was able to deduce the existence of the loop.
of approximately the same complexity as both the demonstration example and the student test problem. We compared POIROT’s performance on the ten problems to the performance of the middle-scoring 40 human subjects (trimmed mean) on the student test problem.

**Statistical Results**
The battery of metrics utilized a distinction between large scale activities, such as selection of a specific web service which we called *choices*, and smaller scale activities, such as selection of parameters to web services, which we called *steps*. The metrics were: accuracy of steps and choices, coverage of observed steps and choices, and goal achievement. The metrics were scored in the same way by an automated program that scored both POIROT and the student subjects. The automated scores were normalized to a range of zero to one hundred.

<table>
<thead>
<tr>
<th>POIROT</th>
<th>Step Coverage</th>
<th>Choice Coverage</th>
<th>Step Accuracy</th>
<th>Choice Accuracy</th>
<th>Goal Achievement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Score</td>
<td>79.65</td>
<td>85.35</td>
<td>82.35</td>
<td>92.90</td>
<td>96.25</td>
</tr>
<tr>
<td>Novice Trimmed Mean</td>
<td>53.58</td>
<td>56.80</td>
<td>76.13</td>
<td>70.60</td>
<td>77.80</td>
</tr>
<tr>
<td>% of Novice Trimmed Mean</td>
<td>149%</td>
<td>150%</td>
<td>108%</td>
<td>131%</td>
<td>124%</td>
</tr>
<tr>
<td>Confidence (t-test)</td>
<td>99.4%</td>
<td>99.4%</td>
<td>99.4%</td>
<td>99.4%</td>
<td>99.4%</td>
</tr>
</tbody>
</table>

Table 1 shows the results for POIROT, the results for a trimmed mean of the student subjects, POIROT’s percentage relative to student performance, and the confidence that POIROT met or exceeded student performance. The trimmed mean was obtained by dropping the three students with the highest goal achievement scores, and the three students with the lowest goal achievement scores. Since student performance tended to cluster at higher performance, the effect of trimming the mean was to provide POIROT a more difficult target, and to reduce the number of human subjects to forty. The confidence statistic was obtained by using a one tailed T-test with unequal variance. As can be seen, POIROT performed at 124% of the student goal achievement score, which measured the effectiveness of the resulting evacuation plan.

**RELATED WORK**
POIROT’s architecture is based on the notion of multi-strategy learning [17, 7]. It provides an environment where multiple learning algorithms can be effectively combined. Michalski provides a taxonomy of learning methods that includes inductive, deductive, and analogical algorithms, all of which apply to POIROT. POIROT also makes heavy use of the general idea of learning goals [14, 8, 12, 22] as a means of organizing these learning activities.

Our learning of task hierarchies is related to methods for learning macro-operators (e.g., [11, 18]), in that they explicitly specify the order in which to apply operators, but macro-operators do not typically support recursive references. Recent learning work in programming by demonstration such as [Lau01] is also related to our approach. Here, text editor macros are learned by reasoning about how to generalize examples using version space algebras. Most if not all of these efforts do not deal with complex actions that directly consume the products of other actions.

There is a large body of work in workflow/process mining. Most of these [1, 6, 2], can discover non-sequential workflows but they cannot handle workflows containing more than one node per action. Among the existing work some [5, 10] employ grammar inference techniques to find the target workflow. Herbst and Kragiannis [10] use Bayesian model merging with the aim of maximizing the likelihood of the model.

The major difference between POIROT most other existing work on workflow mining is our explicit use of I/O dependencies as distinct from precondition/effect dependencies to discover the context of each trace element and use this information to discover the control flow. POIROT uses both. Many other algorithms use neither. Another distinction is that our approach does not require the target workflow to have unique action nodes as most of the others do. WIT creates distinguished nodes based on IO similarity. Furthermore WIT can detect loops and generalize from just one example trace, which is not the case for approaches based on probabilistic likelihood [10].

WIT can also decompose workflows using a simple structure-driven approach which is significantly different from the taxonomy-based approach described in [9]. Although WIT’s approach does not involve search for the best possible decomposition in terms of grouping the actions that are used for achieving a common goal, but DISTILL does.

**CONCLUSIONS**
The POIROT project is now in the third of four year-long phases. The overall objective is the development of an architecture and suite of components that draws on and combines the strengths of a collection of machine learning approaches to solve the difficult problem of learning complex procedural models from a single demonstration. We have already demonstrated near human level performance on some aspects of simple medical evacuation problems, and our goal for this phase is to address a number of issues with regard to uncertain choices and robustness to local failures, utilizing a greater degree abstract knowledge of resources, choice mechanisms, and physical constraints.
ACKNOWLEDGMENTS

We gratefully acknowledge all of the members of the POIROT team, but give special thanks to: Paul Cohen, Marie desJardins, Chris 13, Yolanda Gil, Robert Goldman, Pat Langley, Michael Littman, Drew McDermott, Karen Myers, Tim Oates, Jude Shavlik, Katia Sycara, and Manuela Veloso and their colleagues for numerous discussions and contributions to this effort. We also thank the rest of the BBN team and especially: Brett Benyo, David McDonald, Michael Atigetchi, Karen Haigh, Ray Tomlinson, for all of their hard work.

We thank the many people involved in the POIROT project for their contributions. This project is supported by DARPA IPTO under contract FA8650-06-C-7606. Approved for Public Release, Distribution Unlimited.

REFERENCES


