Web Intelligence and Agent Systems: An International Journal 5 (2011) 1–5
IOS Press

Learning Web-Service Task Descriptions from Traces

Thomas J. Walsh [a,*], Michael L. Littman [b] and Alexander Borgida [b]
[a] *Accenture Technology Labs*
161 N. Clark St.
Chicago, IL 60601 USA
E-mail: thomasjwalsh@gmail.com
[b] *Department of Computer Science*
Rutgers University
110 Frelinghuysen Rd.
Piscataway, NJ 08854 USA
E-mail: {mlittman, borgida}@cs.rutgers.edu

Abstract. This paper considers the problem of learning task specific web-service descriptions from traces of users successfully completing a task. Unlike prior approaches, we take a traditional machine-learning perspective to the construction of web-service models from data. Our representation models both syntactic features of web-service schemas (including lists and optional elements), as well as semantic relations between objects in the task. Together, these learned models form a full schematic model of the dataflow. Our theoretical results, which are the main novelty in the paper, show that this structure can be learned efficiently: the number of traces required for learning grows polynomially with the size of the task. We also present real-world task descriptions mined from tasks using online services from Amazon and Google.

Keywords: machine learning, web services, sample complexity, apprenticeship learning

1. Introduction

The growing availability of web services (interfaces for programs to send and acquire data from web sites) has revolutionized the way applications interact with the World Wide Web. Although in practice most services are chained together manually, researchers have embraced the problem of automatic service composition, especially within the Semantic Web and Web Service communities. Essential to almost all of these approaches, especially ones based on AI [6,14,15,22], is some formal description of the *syntax and semantics* of a service. Unfortunately, such formal descriptions (especially of semantics) are rarely encountered in practice. In addition, even if semantic descriptions are provided, when services come from multiple providers they are almost always incompatible. Therefore *learning* web-service descriptions from experience, rather than relying on hand-coded descriptions or background ontologies [12,11] is a natural and pragmatic approach for modeling web-service behavior.

In this work, we consider a machine-learning approach to build descriptions of how the inputs and outputs of services are related, both for a single service and for a sequence of services comprising a task (such as looking up flights and then booking the one with the minimum price). The inputs to this algorithm are XML documents containing instances of the inputs and outputs of each service in the task. Such *traces* of services in a task can easily be collected in practice. The system uses this data to refine its hypothesis about the *structure* and *meaning* of the services, specifically focussing on semantic relations between concepts in the dataflow. We have chosen this (essentially supervised)

*Corresponding author. E-mail: thomasjwalsh@gmail.com

1570-1263/11/$17.00 © 2011 – IOS Press and the authors. All rights reserved

learning protocol because it allows our system to learn the descriptions of service-based tasks simply by observing (perhaps completely non-technical) users naturally performing a task. For instance, if we are training the service to search and book flights using services like the one in Figures 1 and 2, the system could be trained simply by watching a person perform their normal search and purchase routines, even if they can't perfectly describe their methods.

More specifically, we use an apprenticeship learning [1,28] protocol where agents attempt to execute services to complete a task, and when they fail to do so (as judged by a human observer or some fixed rules), a human generated trace demonstrating the task is provided, either from stored examples or from a human supervisor. Again, the users here only need to understand how to execute the task using their normal service interface. They do not need any special training on how the machine learning algorithm works, nor any knowledge of semantic modeling languages or formal notation. In this way, the supervisor acts as a *teacher*, who can step in when the system make an incorrect prediction or gives the wrong inputs to a service, and either show it how to complete the current task or give it a similar task from stored traces where the situation was handled properly.

This approach stands in contrast to other efforts in aiding technical [16] and non-technical [18] users to encode a control flow (such as the one in Figure 2) through service-composition interfaces. Unlike these methods, which rely on the user being able to use this software *and* have perfect knowledge of their dataflow, our system can learn service descriptions simply by watching people use the services as they always do (perhaps not even realizing they are calling services), even if they are unable to perfectly describe the reasons they use the services in a certain way. In cases where users feel confident using service composition interfaces, they could of course be used to bootstrap the models used in this paper.

Our learning problem also differs from previous efforts that used existing ontologies to learn "data semantics" of service arguments. That is, as noted in [21], and in the design of a number of languages for describing services and their semantics (e.g., WSDL-S, METEOR-S [20]) there are two kinds of semantics one needs to attach to a basic syntactic service description: (i) "data semantics" about the arguments (usually concepts from some ontology), and (ii) more traditional "functional semantics" on the relationship between inputs and outputs, and the outcome of each service. Machine learning has been applied to the first kind of semantics when an ontology is available [11,5,12], and while this is an essential task, we focus here on the orthogonal task of learning the functional semantics of services.

Our work can also be seen as complementary to work in the planning community on finding sequences of web service calls that will achieve a goal [6,14,15]. All of these works considered the case where the model of each service (how its inputs and outputs are related) is given. In contrast, our work is about building such models from data (learning rather than planning). However, we note that unlike many operators used in the planning community, the semantic descriptions learned by our system are task specific—their semantics are only valid within the context of the learned task. For instance, if a task involves buying the cheapest item on a list, a semantic link of "min" might be learned from the "price" output of a search service to the input "payment" of a purchasing service, even though this link is not necessary in all tasks. Our approach sacrifices the full generality of task-independent operators and the expressiveness of complex workflows for tractability and applicability to naturally occurring tasks. We discuss some challenges of resolving this conflict by mining more task-independent operators in Section 7.3 and integrating more expressive workflows (including those that might model transaction failures and exception handling) in Section 8.

Because our approach is data driven, we are particularly concerned with the *sample complexity* of learning, a theoretical quantification of how much interaction is required with an environment to build an accurate model. The theoretical sample complexity results proved in this paper give an upper bound on the number of times the teacher needs to "step in", and the empirical evidence we have collected on real-world applications indicates the number of interventions needed to fully learn a task is usually far smaller than these worst case bounds.

The machine-learning algorithms in this work can be viewed as novel instantiations of classical ideas, but their application to this domain requires many nuanced and innovative design choices. Moreover, the formulation of task-description mining as a machine-learning problem yields theoretical results showing the problem is tractable and our empirical studies show the approach is effective in several natural real-world problems. This novel approach opens the door to a new way

of looking at web-service descriptions that is not beholden to pre-packaged or hand-tooled semantics.

In the next section, we formally define the web-service task-learning problem and our *Task Graph* representation. The definitions and descriptions are meant to solidify the problem, though we will often refer to the flight booking services and task in Figures 1 and 2 for intuitive explanations. After presenting a basic learning algorithm for this problem, we show how to learn in the presence of nested list structures, missing elements, and selections of elements from lists. Throughout, we report results on the sample complexity of our approaches in the *mistake bound* paradigm [13]. After the theoretical study of the learning problem, we give a brief overview of the complexity of reasoning with Task Graphs as applied to the *Input-Selection* problem, where an agent must choose the correct inputs for each service in a task. Then we give examples of service descriptions mined by our algorithm from Amazon Web Services (AWS)[1], the Google Data API, and a task that uses services from both providers. This final example showcases one of the more promising benefits of our machine-learning approach: because our approach does not rely on any external ontology, there is no need to reconcile incompatible or missing semantic descriptions from the two different providers. Finally, we discuss extensions of our algorithm and representation for modeling stochasticity, changing services, and a translation to more traditional planning operators.

2. Terminology and Representation

Intuitively, the learning problem we consider is one where an agent must complete a sequence of web-service calls to successfully accomplish a task, such as using three services to look up a person's vacation destination, look up the flights to this location, and buy the cheapest ticket. As examples, Figure 1 shows several representations of an individual service call and Figure 2 shows an example dataflow between multiple services. Whenever the agent's model leads to a mistake with respect to the task when provided new inputs, a *teacher* steps in and shows the correct service behavior for that particular instance. This intuitively simple interaction is formalized in the following sections, followed by a summary of some of the more vexing learning issues considered in this paper.

[1] http://aws.amazon.com/

2.1. Formal Problem Description

We begin by formally defining our *task graph* representation. Throughout this section, we refer to Figures 1 and 2 for examples of the terms defined. Since web services communicate via XML documents, we describe the semantics to-be-learned of services starting with XML DTDs describing their inputs and outputs[2].

To simplify the presentation, throughout this paper we eliminate XML attributes by simply treating them as child elements. As usual, the declaration of elements in DTDs can be represented using regular expressions over an alphabet defined by the sub-element/tag names. For instance, the expression for "Res" (to be used in Figure 1) would be (NumRes Flight$^+$ MaxPrice MinPrice). Though learning general regular expressions would be intractable, studies have shown [2] that the expressions representing XML element declarations in 99% of XML DTDs on the web are from a restricted language, *chain regular expressions* (CHAREs), where element names can only repeat in a list, and quantifiers, such as $^+$ and * are only applied to flat structures such as symbols or disjunctions of symbols, without any nesting. In this work, we further assume services do not have disjunctive elements (none of our real world experiments contained these), leading to the restricted class of non-disjunctive CHARE defined below.

Definition 1. *Non-disjunctive CHARE:*

- *A* non-disjunctive Chain Regular Expression *(non-disjunctive CHARE) is a string ρ of the form $Q_1 \ldots Q_n$, where each Q_i, $1 \leq i \leq n$, consists of a symbol a_i from alphabet Σ, followed by* at most one *"annotation" symbol from the set {$^+$, *, ?}, and each a_i may only appear* once *in ρ. Therefore, a^*bc^+ is a non-disjunctive CHARE, but a^*ba and $(ab)^+$ are not.*
  Non-disjunctive CHAREs form a subclass of extended regular expressions, where neither disjunction nor nesting is allowed.
- *A* non-disjunctive CHARE element declaration *is a DTD element declaration* `<!ELEMENT A( ρ )>`*, where ρ is a non-disjunctive CHARE.*

[2] Note that the inputs to our learning system will be XML document instances, from which such DTDs, among others, will need to be learned.

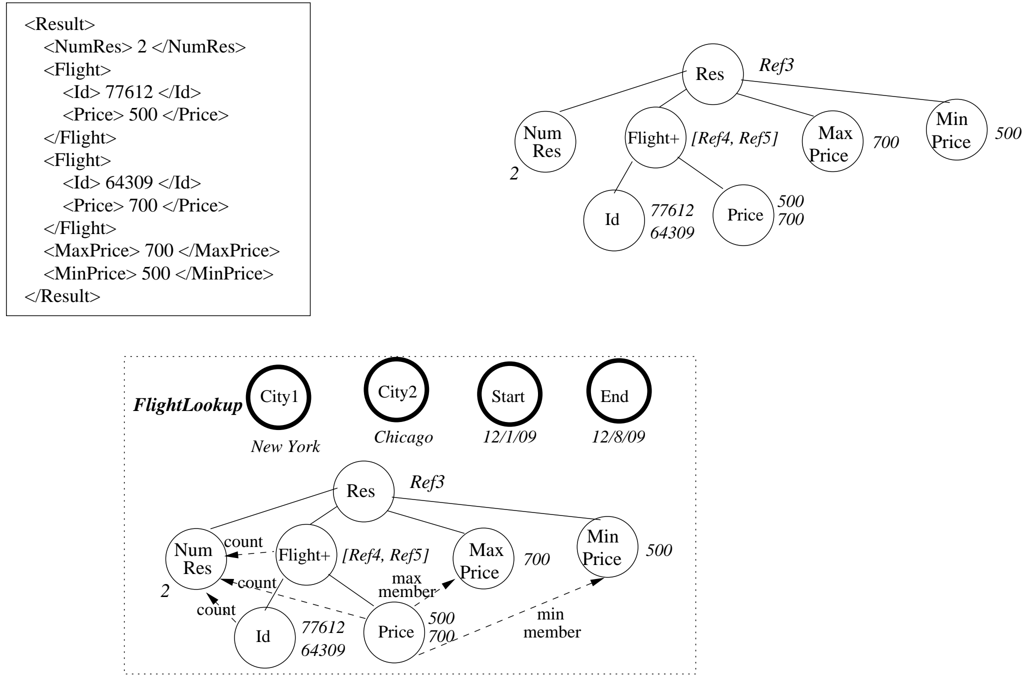

Fig. 1. The output of a ***FlightLookup*** web service in XML form (top left), the associated XML structure tree instance (top right) and the full service instance represented as a graph (corresponding to Definition 9), complete with semantic edges (the dashed edges) representing relations that should hold between instances corresponding to a pair of nodes. In the bottom graph, objects appear next to their corresponding nodes. When multiple objects map to the same node, they are grouped either as a list (the horizontal tiling for Flight$^+$) or as a Sublist (the vertical tiling for Price) based on Definition 4 . For nodes with multiple parts, our implementation hashes all the text (including tags) below the elements so that we can check for exact equivalence between structures. This results in the "Ref" objects in the figures in the paper, which are simply shorthand for these concatenated strings.

– *A* non-disjunctive CHARE DTD *is a DTD where every element declaration is a non-disjunctive CHARE element declaration, and the DTD is* non-recursive.

We represent such DTDs by trees in the following way, by essentially unfolding the element declarations in context.

Definition 2. *Given a non-disjunctive CHARE DTD, its* non-disjunctive CHARE Structure Tree $\langle N, E\rangle$ *(henceforth called an "XML Structure Tree") is a node-labelled graph consisting of a set of nodes N, each labeled with the name of a DTD element. The tree is constructed recursively by starting with a node n_0, labelled by the root element of the DTD. For every node x with label A, if the DTD has declaration*
<!ELEMENT A (Q_1, ..., Q_k)>
then children nodes $y_1, \ldots, y_k$ are added to N (with labels $a_1, \ldots, a_k$), and x is linked to these child nodes by edges added to E. Moreover, for each Q_j in the element declaration that has an annotation $^+$, ?, or * , the corresponding node y_j is annotated *by $^+$, ?, or * respectively. Formally, an XML Structure Tree therefore has an associated* annotation partial function $\varphi : N \mapsto \{^+, ?, ^*\}$.[3] *Nodes annotated with $^+$ or * will be called* list nodes*, while nodes annotated with * or $^?$ will be called* optional nodes.

Ignoring the text outside of the circles, the top right of Figure 1 represents an example XML Structure Tree for the non-disjunctive CHARE DTD with declarations
<!ELEMENT Res (NumRes, Flight$^+$, MaxPrice, MinPrice)>
<!ELEMENT Flight (Id,Price)>

We emphasize that though the above definition starts from a DTD and builds a tree, in our setting the tree will be learned from documents – the learning algorithm is not given a DTD.

An XML Structure Graph is obtained from an XML Structure Tree by also labeling edges as follows:

[3]When drawing the tree, we leave the annotation as part of the name inside the circle representing the node, together with the label.

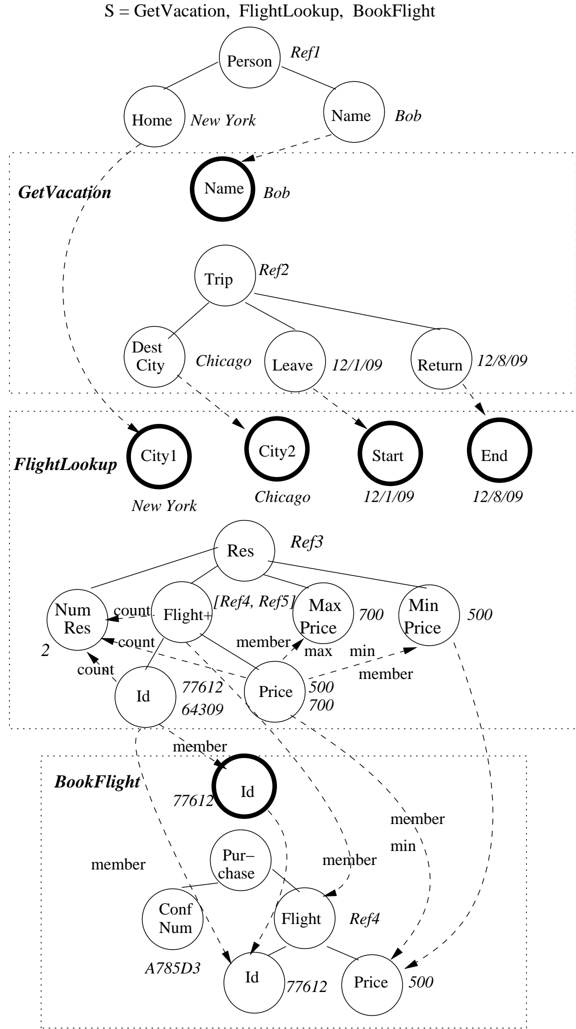

Fig. 2. A task graph instance for an agent looking up a person's vacation information, searching for flights, and booking the cheapest one. An XML trace provides the structural (solid) edges, but not the semantic (dashed) edges, which must be learned. Inputs to services are shown with bold circles.

Definition 3. *Given an XML Structure Tree* $\langle N, E, \varphi \rangle$*, the corresponding* XML Structure Graph $G_{str} = \langle N, E', \varphi, \Lambda_{str} \}$ *has the same set of labeled nodes* N *(complete with annotations), but for every* unlabeled *edge* (x, y) *in* E*, there are two labelled edges in* E'*, with label values from the set* $\Lambda_{str} = \{part, whole\}$*: edge* (x, y) *labelled* $part$*, and edge* (y, x) *with label* $whole$*. These are called* structural edges*, and they are intended to capture the intuition that in a non-recursive XML tree, the children are "parts" of the parent "whole".*

In diagrams, we continue to show each part/whole pairing a single solid edge for readability, because of the regularity of the edge labeling and creation.

We now consider the instantiations (corresponding to documents) of these structures (which are schematic descriptions). We continue with the XML Structure Tree shown in the right upper corner of Figure 1 and the XML document at the top left of Figure 1. Although there are well-known ways to build trees for XML documents (e.g., the so-called DOM structure) these are not useful for us because we want to associate the values in the document with the nodes of the XML Structure Tree directly (since we will be trying to learn this DTD-like model). This is easy for elements whose tag occurs only once, at a leaf, such as NumRes: we just put the PCDATA value 2 beside the corresponding node in the XML Structure Tree instance. If an element E is repeated (in the XML Structure Tree this would be a node annotated with * or $^+$), then we can simply gather the objects hanging from the DOM tree for it into an associated *list* (e.g. [Ref4, Ref 5] besides Flight$^+$), where Ref4 and Ref5 are canonical representations of the DOM trees for the different instantiations of Flight.[4]

The problem gets complicated with children of repeated internal nodes, such as Id or Price under Flight node, since there is only one Flight node in the XML structure graph, yet both the Flight and its children are repeated. The idea we follow is to superimpose the successive occurrences of element Flight for all nodes beneath it until another list is reached (we provide further details on this below). This leads to nodes such as Id having associated so-called *sublists* $\langle 77612, 64309 \rangle$. Note that our restrictions on non-disjunctive CHARE are essential in allowing such structures to be built because of the reduced variability.

The figures in this paper showing task graphs actually all show *task graph instances* because we feel they are easier for readers to follow. Here we formally define "instances" of the structures described so far and, where necessary, show how the instances are populated from the original XML documents. We begin by defining *objects* which are ground instances of the XML element represented by each node in the XML structure graph.

[4]Actually, for nodes with parts, we hash all the text (including tags) below the corresponding elements so that we can check for exact equivalence between structures. The "Ref" objects in the figures in the paper are simply shorthand for these concatenated strings.

Definition 4. *An* object *is either* atomic *– a grounded element from an XML document (*`#PCDATA` *or a reference to all the* `#PCDATA` *below this element) , or a* sequence *of objects. A* sequence *is either a* list *of atomic objects* $[a_1, a_2...]$ *if an element repeats multiple times at a given node in the parse tree of the document, or a* sublist *if a "whole" element above A repeats – denoted* $\langle a_1, a_2...\rangle$*). Note that if both cases hold, one can have a sublist of lists (e.g.* $\langle [a1, a2], [a3, a4, a5], [a6]\rangle$*).*

In our diagrams for structure instances, we place objects (e.g. *New York*) beside the corresponding nodes. Therefore, as explained above, the Flight$^+$ node in Figure 1 has a list associated with it (horizontally tiled objects in the diagram) while the Price node under it is associated with a sublist (vertically tiled objects in the diagram) because every flight in the list has an associated price.

Definition 5. *An* XML Structure Graph Instance *is a pair* $\langle G_{Str}, \mathcal{I}\rangle$ *where* G_{Str} *is an XML Structure Graph and* $\mathcal{I}$*:* $N \mapsto O$ *maps each node in* G_{Str} *to some object in O.*

Given the intuitions from the example above, note that any node with a $^+$ or * annotation can map via $\mathcal{I}$ to a sequence of objects, while nodes annotated with a * or ? may map to an empty sequence of objects. We emphasize that an XML Structure Graph Instance is a schema representation coupled with a mapping to objects obtained from a document, as captured by $\mathcal{I}$. We now outline the rules to be satisfied by $\mathcal{I}$ based on the schema captured in the non-disjunctive CHARE Tree T.

1. If node n with label A in T is a descendant (via part edges) of a list node, then let n_p be the closest such ancestor. Then the ground instances of elements A matching n (by simple parsing) are put into a "sublist" sequence $\langle \alpha_1, \alpha_2...\rangle$ based on which element in the n_p list they are a part of. $\mathcal{I}$ maps n to this sublist, but each α_i is also processed in turn based on the next two rules, as it may contain several (or no) instances of A in a list.
2. If an instance of element name A corresponding to node x in the document is optional and missing in the document, or if this is the case for any ancestor of x in T, an empty object [] is associated with x (or for the corresponding α_i, if the previous rule was used).
3. If instances of element A repeat contiguously in the XML document (the node x will have annotation $^+$ or *), these ground instances are put into a list $[\alpha_1, \alpha_2...]$. If the "sublist" rule above already partitioned the instances of A, the list is associated with a single α_i in the sublist, resulting in a "sublist" of lists.

Examples of missing elements and lists inside sublists appear in the more complicated Figure 3 (Stops* is optional and both Stops* and Eatery$^+$ have sublists of lists),

This takes care of modeling the syntax of valid documents as well as mapping XML documents to objects and then to XML Structure Graph Instances. But we are ultimately concerned with modeling semantics based on mathematical relations, so we now define a set of relations over objects that we will then use to model semantics in an extension of the graph structure constructed above.

Definition 6. *A mathematical relation* m *is a binary relation between pairs of objects,* $m : O \times O \mapsto \{true, false\}$*. We use* $\mathcal{M}$ *to refer to a finite set of such mathematical relations.*

This set can be any arbitrary collection of binary relations and our theoretical analysis provides bounds based on the cardinality of this set. However, to make our examples more concrete, we take a cue from database theory [9] and import five basic functions, which we turn into binary relations: *min*, *max*, *sum*, *average*, and *count*, along with an *identity* relation to check equality of objects. This set seems reasonable since many web services are simply wrappers around database operations. Since we will be dealing with lists (and sublists) to model sequences of objects as described above, we will also consider the basic list relations *member* [5], *first*, and *rest*, where the latter two are used to represent list construction and the identity relation for both lists and singletons. Throughout the paper, we will extend this basic $\mathcal{M}$ to model increasingly complex tasks, including relations for modeling selection of objects in Section 4.3 and modeling relationships between dates in Section 6.1.

We now introduce semantic edges based on these relations.

Definition 7. *A* semantic edge e_{λ_m} *is a labeled edge between two nodes* n_1 *and* n_2 *in a graph, with label*

[5] *member*(L, e) for list L and element e in this work should be interpreted as "Has-Member" (e.g. *member*([1,2,3], 1)).

λ_m, where $m \in \mathcal{M}$. (Such an edge is intended to mean that for any objects o_1 and o_2 instantiating nodes n_1 and n_2 (i.e. $\mathcal{I}(n_1) = o_1$ and $\mathcal{I}(n_2) = o_2$), $m(o_1, o_2)$ must hold true.) A semantic edge instance *is just a semantic edge connecting two instantiated nodes n_1 and n_2 (as in Definition 5). A semantic edge instance is* semantically valid *if and only if $m(o_1, o_2)$ is true for $\mathcal{I}(n_1) = o_1$ and $\mathcal{I}(n_2) = o_2$.*

Semantic edges represent task-specific relationships such as "the City2 input to ***FlightLookup*** in Figure 2 should be filled with a DestCity instance from ***GetVacation***". In this example, City2 and DestCity are nodes (with no annotations), *Chicago* is an object, and this particular edge is labeled $=$ because m is the equality relation (in our diagrams we omit the equality label for readability, and use dashed edges to represent semantic relationship, in contrast to structural (solid) edges). Unlike structural relations, we cannot assume these semantic relations are provided to us directly from the XML structure. However, each XML instance does give us information about what relations might consistently hold true. To learn these relations, we assume we have a set of common mathematical relations ($\mathcal{M}$ above), all of arity 2, whose semantics are known and can be easily checked[6].

A graph with structural and semantic edges is called an *SS-graph*.

Definition 8. *An* SS-Graph *is a labeled directed graph $G_{SS} = \langle N, E, \Lambda \rangle$ containing the nodes from an XML structure graph G_{str} and has edges $E = E_{str} \bigcup E_{sem}$ where E_{str} contains all the edges from G_{str}, and E_{sem} is a set of semantic edges that must hold true in all instantiations . Λ is $\{\lambda_m | m \in \mathcal{M}\} \cup \{part, whole\}$.*

We can now formally define a service in terms of its inputs and outputs, though we do not yet allow for semantic connections between these components.

Definition 9. *A* service *s is a triple $\langle \eta_s, G_I, G_O \rangle$, where η_s is simply the name of the service, and G_I and G_O are each SS-Graphs , where G_I intuitively represents the inputs and G_O represents the outputs.* [7]

[6]The theoretical efficiency results of this paper generalize to relations of constant arity by building the corresponding hypergraph.

[7]Equivalently, a service can be defined in terms of XML elements that form its inputs and outputs (E_i and E_O) along with a set of relations (R_s) that must hold within their graphs. However, the equivalent graph structure is easier to visualize and has a well defined construction from data (Definition 2) so we use this representation here.

An example of a service is the ***FlightLookup*** box at the bottom of Figure 1, which contains a 4-node input graph and 7-node output graph. Such a "message generation" definition of a service follows the descriptions of services often seen in the Web Service Composition community [14].

Since there are often semantic relationships between the inputs and outputs of a service, we define a similar structure that captures these relationships.

Definition 10. *A* service transformation *is a pair $\langle s, \hat{E} \rangle$ where s is a service, and $\hat{E}$ is a set of semantic edges that link nodes of G_I to G_O, thereby forming a larger graph G_s. We assume that the service's behavior is deterministic given the inputs, and that the semantic relations encoded in $\hat{E}$ must always hold for any invocation of service s (though other relations can hold in any single instantiation).*

An example of such a structure is the **BookFlight** box in Figure 2 where the input graph (a single node) has a semantic link to the output graph.

The goal of our learning algorithm will be to model a sequence of service transformations, including relational links between transformations. This target hypothesis is called a *Task*, T^*, and is defined as follows.

Definition 11. *A* task *is a pair $\langle S,R \rangle$ where S is a* ***sequence*** *$\eta_1 ... \eta_m$ of service names that induce a partial ordering $\preceq_S$ over the nodes and objects in the corresponding service transformations: $x \preceq_S y$ for every node x in η_i and every node y in η_{i+1}, $1 \leq i \leq m$. R is a set of triples $\langle n_1, n_2, m \rangle$ for nodes $n_1 \preceq_S n_2$ and $m \in \mathcal{M}$, and where $m(\mathcal{I}(n_1), \mathcal{I}(n_2))$ must be true for all $\mathcal{I}$.*

Notice that in a task, semantic relations include not only those from each service transformation, but also relations between elements from different services. Intuitively, S represents what is called the "control flow" in process mining [29]– a sequence of service calls necessary to complete the task (see the top of Figure 2). R (representing the dataflow) encompasses relations between objects in service graph instances G_i and G_j, where $G_i \preceq_S G_j$, including relations within (i) the same graph (a service), (ii) relations between the inputs and outputs of a service (a service transformation), and (iii) those between graphs associated with different services (task specific relations).

Internally, we will represent a web-service task using a *task graph* as defined below. The edges in the task graph include all the structural edges from the individual services, but also include directed edges for

the semantic relationships between objects. Intuitively, the semantic edges, including those associated with the *identity* relation (shown as unlabeled dashed edges), maintain a valid hypothesis over semantic relations between objects, including which outputs link to which inputs and the semantic relationships between objects in general.

Definition 12. *A* Task Graph *is an SS-Graph representing a task. The (potentially annotated) nodes in the graph are all of the nodes in the SS-Graphs in the service transformation sequence S that is part of a task. The edges correspond to the union of all the edges in those graphs as well as inter-service semantic edges (type iii above) corresponding to the semantic relations R in the task T^*, as defined above.*

Instances of SS-Graphs, service graphs, and task graphs are all obtained by associating with them an instantiation function $\mathcal{I}$ coming from the underlying XML structure graphs. An example of a Task graph instance is shown in Figure 2.

2.2. Learning A Task Graph

We are concerned with learning task graphs from *traces* of the task (task instances). Traces do not necessarily contain direct information about the semantic relations; instead they record a sequence of service instances that give clues as to these semantic relations. For most web services, traces can be obtained as sequences of XML documents exchanged by the client and the service. For REST services (where the service inputs are encoded as in the URL) the inputs can normally be converted into a G_I with no structural relations using simple parsing rules (splitting variables based on & and splitting names and values based on =, as we did in our experiments with Google services). These traces can be produced by users that are experts in completing the task (such as a travel agent with a specific service interface) without any knowledge of formal notation or the learning algorithm used by the agent. Users just have to know how to call the services (using their normal tools) to complete the task correctly. The protocol (described more formally below) alternates between the agent attempting to complete the task, and then the human (or some fixed rules) judging whether or not a mistake has been made, and if so, providing a trace to help correct the mistake. Thus, the user is acting as a *teacher* to demonstrate how the task is performed. With the help of the above definitions, we can now define the task learning problem, which will be our main consideration throughout the rest of this work.

Definition 13. *A* Task Learning Problem *proceeds as follows. Initially, the learning agent is provided with the names of the services to be called (S from the task definition), the set of relations to be considered $\mathcal{M}$, and an initial task instance (trace) τ_0. The* task graph learning problem *then occurs in a series of* ***episodes****. At the beginning of each episode, an initial SS-Graph G_0 is provided to the agent. The agent must then for each successive service in S :*

1. *Provide the instances of the input elements (a semantically valid SS-Graph G_I). The reasoning behind this* input-selection problem *once a model has been constructed is discussed in Section 5.*
2. *Make correct predictions about what semantic relations will hold with respect to the true (but unknown) Task T^*. When possible, this may involve predicting the actual instantiations of nodes, as in the input-selection problem above, but the agent may also make a more generic prediction, simply stating the relationship between two nodes (e.g. Node $n1$ will contain the maximum value from the (not yet instantiated) list in node $n2$).*
3. *Predict the structure of each service instance, including annotations. Specifically, the agent must identify all possible nodes n and structural edges e_{struct} in the service instance as well as whether these nodes can be optional, lists, or both. However, the exact instantiations (the values assigned to each node) do not have to be predicted, except for the inputs as specified above.*

If during an episode, the agent errs in any of these (as judged by a teacher) and *if the current task instance refutes the agent's prediction, this is counted as a* ***mistake*** *and the agent is provided with the task instance (trace τ_t) showing a full run of the episode and all the instances.*

For instance, in Figure 2, the agent is initially given the "Person" structure at the top. The agent must then make the correct sequence of service calls, also using the correct inputs for each service from the previous outputs and making correct predictions about the data produced, as outlined above. An agent that has learned the task graph in Figure 2 can predict that a ***FlightLookup*** will produce a list of at least one Flight (from the annotation $^+$ on Flight) and that the MaxPrice node will contain the maximum value from the

Price node. If the agent makes any mistakes (Definition 13) as judged by the human teacher, it receives a trace (task instance τ) as feedback. This feedback can either be a correct trace for the previous episode (as defined above and considered throughout this work), but more generally could be any example that will correct the misconceptions that led to the mistake (for example a stored trace from a similar instance).

We consider the efficiency of task learning in the *Mistake Bound* paradigm [13].

Definition 14. Efficient Task Learning *occurs if the number of mistakes (as defined at the end of Definition 13) it makes over its lifetime is bounded by a polynomial function of the input parameters* $\{|G_I|, |G_O|, |S|, |\mathcal{M}|\}$*, where* $|S|$ *is the length of the sequence of services to be called and* $|G_I| = \max_j |G_{Ij}|$ *for* $j = 1 \ldots |S|$*, and similarly for* $|G_O|$*, and with* $|G|$ *being the number of nodes in graph* $|G|$.

In practical terms, this constraint ensures we can train agents to perform complex tasks involving web services with a number of examples that scales polynomially with the size of the task schematic. This efficiency is crucial for any practical realization of this system as traces of specific tasks, while not necessarily scarce, certainly will be limited. Also, we note that the theoretical bounds we prove here will rarely be met in practice as they are done in a worst-case analysis. Our experimental results (Section 6) indicate the number of mistakes for large-scale tasks will actually be quite small.

Finally, we note that other frameworks could also be used to bound the sample complexity of similar tasks, but we chose mistake bound because it can be used in the deterministic online case, where it only counts the number of mistakes made (and therefore traces needed) and does not demand that these mistakes are all made during any particular phase of learning. This is the most natural framework for "over the shoulder" teaching framework, where a teacher needs only to step in at times when the learning agent actually makes an error. Also, recent work on the sample complexity in apprenticeship learning (a protocol very similar to ours) [28] has shown that mistake-bound learnability of a class is sufficient for its efficient learnability in the apprenticeship setting.

2.3. Key Learning Problems

While the presence of traces certainly makes the web-service task learning problem easier than a completely unsupervised approach, a number of non-trivial learning tasks remain. Here, we provide a sketch of the key challenges in the web-service task-learning problem.

- **Ambiguity** - A single (or even multiple) traces may not settle all the semantic relations between objects. For instance, if two lists are visible to a service (A=[1,2,3] and B=[3,4,5,6] and the service produces "3": was that max(A), min(B), or length(A)? Further traces are required to determine the correct pairing (if there is one).
- **List Structures** - Lists of objects are ubiquitous in web services. Figures 1 and 2 show such output with the list of possible flights. Detecting and modeling lists, including learning when lists are potentially empty, is an important portion of the overall task-learning problem. While WSDL documents or other syntactic schemas often indicate which elements may repeat, we show in this paper that under very common assumptions, the presence of lists and missing elements can be learned as well. This makes our learning algorithm more practically robust.
- **Sublists and Selection** - When lists contain non-primitive structures, the parts of the elements in the list (e.g., the Price of a Flight from the flight list in Figure 1), form a *sublist* as covered in Definition 4. The elements of a sublist are not grouped together in the original XML document, but, we may need to consider them as a group to learn some semantic links. Additionally, the dataflow from some nodes may best be expressed in terms of the grouping induced by a "whole". For instance, in Figure 3, when a Flight is chosen, its "Stops" list is copied over, so the dataflow should capture the fact that these stops all belong to the same flight, and if possible, the reason for this selection.

3. Simple Task Learning

Before we handle the general form of the learning problem presented in Definition 13 , we consider a simplified web-service task-learning problem with a number of assumptions. We assume lists (e.g., the Flight list in Figure 1) are not nested inside one another and can never have length 0 (no missing elements). We further assume that any time multiple parts of an object that came from a list appearing in a later SS-Graph, the entire structure (not just a few parts) will appear in the later graph. For example, the entire Flight ob-

ject appears in the output graph of ***BookFlight***, not just the Price and Id. In subsequent sections, we will relax all of these conditions, but we study this "simple task-learning problem" to convey the basics of our learning algorithm.

3.1. Mapping XML to Structure Graphs

Each input (trace τ) to our learning algorithm comes as a series of XML documents showing the inputs and outputs of each service instance. If the syntax of each service (what elements are lists and which ones are optional) is provided through correct WSDL or other documentation, translating these instance documents to XML Structure Graph instances can be done using the procedure outlined in Definitions 2, 3, and 4. However, we consider here the more general situation where this documentation is not provided, and hence the translation from XML documents to a task graph instance requires learning the syntax of each service's inputs and outputs (XML structure subgraphs of G_I and G_O). As noted earlier, this syntax (the XML-structure tree) can be represented using a non-disjunctive CHARE (Definition 1) for each XML element. For instance, the expression for "Res" in Figure 1 would be: (NumFlights, Flight$^+$, MaxPrice, MinPrice). With only the traces to work from, these forms must be learned from multiple traces because each trace instance may only provide partial information about whether an element is a list (singletons and lists of length 1 are often indistinguishable) or optional (covered later).

Because of the non-disjunctive CHARE constraints, when we see two "Flight" elements under a "Res" element, we can infer it is a list (Flight$^+$ or Flight*), not a sequence of two Flights, since no duplicate element names can appear in a declaration. Therefore translating XML documents to instantiated graph nodes and *structural* edges in an XML Structure Tree, which forms the backbone of a Task Graph G_T, is straightforward, even in the online-learning case. Each document maps to a specific service $\langle G_I, G_O \rangle$ by the rules in our earlier definitions. That is, each XML element can be represented by a non-disjunctive CHARE (where repeated elements will have a $^+$) and each symbol in the non-disjunctive CHARE becomes a node in the graph. If there is a $^+$ on this symbol and the node in the graph does not yet reflect it, the annotation is added (other quantifiers are considered in the next section). Part-Whole relations and instances are then copied in from the XML. The instances of each node are populated from all the corresponding XML elements. Thus, under the assumptions above, learning the XML-structure trees within the true Task Graph G_T^* is tractable. The efficiency is discussed in the next section and modifications to the structure learning when these assumptions are relaxed are discussed in Sections 4.1 and 4.2. But now we turn our attention to the goal of learning the semantic relations.

3.2. Learning Simple Task Graphs

Our goal is to construct a model of the syntactic and semantic relation in the true task T^* with at most a polynomial number of mistakes. The Task Graph Learning Algorithm (TGLA: Algorithm 1) does so using the Task Graph (G_T) representation by ruling out possible semantic relations between elements based on traces.

Since we are temporarily assuming each XML element appears at least once, the first trace bootstraps all the nodes in the task graph, though $^+$ annotations may still need to be refined, and there are potentially incorrect semantic links. For instance, if the first trace in our flight-booking domain had a person whose name and home city were both "Austin", then both the "Home" and "Name" nodes in the graph would link (through the identity relation) to the "Name" node in ***GetVacation***. This will be problematic if the next episode starts with Bob from New York. Should the agent call ***GetVacation*** with "Bob" or "New York"? In the mistake-bound setting we have considered, when such ambiguity exists the agent can just pick one of the possible choices. If it is wrong, it will receive a trace and be able the eliminate the incorrect link. Even with more complicated semantics that require super-polynomial computation to make predictions (see our overview of reasoning with several different variant of $\mathcal{M}$ in Section 5), it only takes one trace with contrary information to remove a link. Therefore, the mistake bound is on the order of the number of edges that might need to be eliminated from G_T. More formally (a proof is provided in Appendix A):

Proposition 1. *TGLA for Simple Tasks makes* $O((((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ *mistakes in the simple web-service task-learning problem when the target semantics are representable.*

Again, the polynomial sample complexity bound shows that the amount of data needed in the worst case for learning a task graph scales only quadratically with the size of the task in the online mistake-bound set-

Algorithm 1 Task Graph Learning Algorithm (TGLA) for Simple Tasks

1: **Input**: $\mathcal{M}$, initial trace τ_0, service sequence S
2: **Output**: Behavior eventually consistent with task T^*
3: Construct S and $\preceq_S$ exactly from τ_0
4: Extract structure graphs G_I and G_0 for every service in τ_0 as defined in Definition 3.
5: $G_T = \bigcup_{j=1}^{|S|} G_{Ij} \cup G_{Oj}$ //All the nodes and edges of all the structure graphs
6: **for** Every pair of nodes $(n_i, n_j) \in G_T$ where $n_i \preceq_S n_j$ and every $m \in \mathcal{M}$ **do**
7:   **if** $m(n_i, n_j)$ holds for the instances of those nodes from τ_0 **then**
8:     Construct the corresponding edge $\langle n_i, n_j, \lambda_m \rangle$
9:   **end if**
10: **end for**
11: **for** each episode **do**
12:   //Main Learning loop
13:   The agent receives an initial set of instances G_{O0}
14:   Execute each service in S , choosing inputs and making predictions by generating instances that are consistent with the semantic links in G_T
15:   **if** Trace τ received from the teacher (indicating a mistake has been made) **then**
16:     **for** each semantic edge $e = \langle n_1, n_2, \lambda_m \rangle \in G_T$ **do**
17:       Let $\mathcal{I}_\tau$ map nodes in G_T to their corresponding objects in τ
18:       **if** $m(\mathcal{I}_\tau(n_1), \mathcal{I}_\tau(n_2))$ is false (invalid) **then**
19:         Remove e from G_T.
20:       **end if**
21:     **end for**
22:     **for** each semantic node $n \in G_T$ **do**
23:       **if** $\mathcal{I}_\tau(n)$ is a list of objects (see definitions 2 and 4) **then**
24:         Annotate n with a $^+$.
25:       **end if**
26:     **end for**
27:   **end if**
28: **end for**

ting. Such results are important in machine learning as they indicate the tractability of algorithms as their instantiations become larger. Similar bounds are possible (under different assumptions) in other frameworks such as PAC [24] for the batch setting with a distributional assumption on the types of services. Finally, we note our experiments will show that the actual number of mistakes needed to learn many real-world services is actually quite small.

4. Full Task Learning

We now relax the earlier restrictions, allowing nested lists, missing elements, and selection semantics. Each change, leads to increased sample and computational complexity for TGLA, though the respective bounds for all of these extensions remain polynomial. To demonstrate these properties, we introduce a second "Flight Booking" example in Figure 3. This example contains several features that were formerly prohibited, including nested lists, optional elements, and portions of complex structures "selected" from earlier services. The complexity of learning such Task Graphs is considered in the following subsections.

4.1. Nested Lists

Allowing nested lists (as with "Stop*" and "Eatery$^+$" in Figure 3, where each entry in a list of stops has its own list of eateries) requires a change in the entries of $\mathcal{M}$. As a matter of notation, we denote the maximum nesting depth, (which is 3 in Figure 3, but at most $\max(|G_O|, |G_I|)$), as d in subsequent bounds. Once $d > 1$, it is possible for sublists to also contain lists (as with the "Eatery$^+$" node). Hence, the semantic links may become ambiguous (does "member" mean the node contains a single list from the sublist, or is it a sublist of members from each list?). This is rectified by creating 2 versions of each relation $m \in \mathcal{M}$: *list*-m and *sublist*-m.

We use these different forms of the relations (as defined below) to resolve the ambiguity discussed above, though we note that other changes to $\mathcal{M}$ are possible to achieve the same desired effect. Specifically, we can interpret *list*-$m(\mathcal{I}_\tau(n_1), \mathcal{I}_\tau(n_2))$ from line 18 of Algorithm 1 using the following semantics (with $\mathcal{I}$ as shorthand for $\mathcal{I}_\tau$):

- If $\mathcal{I}(n_1)$ is a list instance, and $m(\mathcal{I}_(n_1), \mathcal{I}(n_2))$ is true, then the edge is valid.
- If $\mathcal{I}(n_1)$ is a list instance and $\mathcal{I}(n_2)$ is a sublist of objects $\langle o_1 ... o_n \rangle$ then if $\forall_i m(\mathcal{I}(n_1), o_i)$ is true, then the edge is valid. This is to indicate, for instance, that all the single objects in n_2 (which are

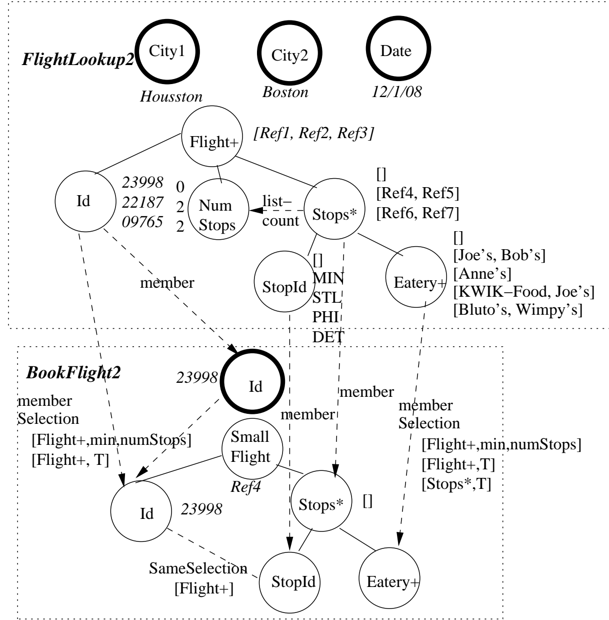

Fig. 3. A partially learned task instance. Only some Selection edges are shown (for clarity). SameSelection links can exist between all the parts of SmallFlight. Further learning (on instances with no 0 or 1 stop flights) could eliminate the extra Selection label ([1,T]) shown for Eatery$^+$.

only grouped because of shared parent structure in the XML) are members of a list from n_1 [8].

- If $\mathcal{I}(n_1)$ is a sublist of lists $l_1...l_n$ (as in the "Stops*" node in Figure 3), n_2 must be instantiated with a corresponding sublist where for each object $\langle o_1...o_n \rangle$, $m(l_i, o_i)$ is true for the edge to be valid. This produces a "mapped" version of m as with the "list-count" edge connected to Stops* in Figure 3.

If none of those cases hold, the edge is not valid. *sublist-m* has the same behavior in the first two cases above (but with sublists in n_1 instead of lists), but in the third case, when n_1 contains a sublist sl of lists as in the "Stops*" node in Figure 3, n_2 must contain an object o such that $m(sl, o)$ is true. For instance, "sublist-count" applied to the Stops* node in Figure 3 would link to a node containing "3", the number of lists in the sublist. As a second example from the more familiar Figure 2, list-count on the Price node would only match another node with up of the sublist [1,1] (price looks like it has two lists, each of length 1), while sublist-count will only match Price to a node containing the single instance 2 (the length of the sublist). This is important for being able to maintain the list of valid relations since lists of size 1 and singletons are essentially indistinguishable in the XML documents. For simplicity, where ambiguity does not exist in our examples, we omit this distinction. This extension only increases $|\mathcal{M}|$ by a factor of 3 and therefore does not affect the bound stated earlier. However, the presence of nested lists gives rise to several difficulties regarding missing elements and instance selection. We now consider these problems in detail.

4.2. Learning with Missing Elements

Sometimes, the results from services can have missing elements. For instance, the non-stop flight in Figure 3 has an empty Stop list (and corresponding sublists). This structure is captured in XML Structure tree nodes using the * annotation for potentially empty lists and the ? annotation for potentially missing singletons. We now add these quantifiers to the possible annotations of the nodes in the task graph G_T the same way we utilized the $^+$ annotation on the Flight node in Figure 1. As with $^+$, it is possible that WSDL or other documentation provides this information before the task-learning problem begins, in which case no learning about these syntactic forms needs to be done. But we consider here the worst case situation where all we have is traces. In that full learning setting, these new annotations are adjusted in the following ways by TGLA given a trace (these conditions would be checked in the block of Algorithm 1 between lines 22 and 26, with the added condition that the loop at line 22 must now consider nodes that have been added by the new trace.

- If a node $n \in G_T$ has no annotation or $^+$ annotation, but an instance $\mathcal{I}_\tau(n)$ does not appear in the current trace, change the annotation to ? or *, respectively.
- If the trace τ contains a new node n that is not in G_T, create the new node with annotation ? or *, depending on whether $\mathcal{I}_\tau(n)$is a list or not.
- If a node n has no annotation or is annotated with ? and $\mathcal{I}_\tau(n)$ is a list, change the annotation to $^+$ or *, respectively.

Note that once a node has been determined to be optional or a list (or both), its annotation never goes back. We also need to deal with the semantic edges for these nodes, which might exist between optional

[8] Similar relations can be considered when n_2 contains a list, but in our studies the relation mentioned here seems to occur more frequently

nodes and required nodes (the "list-count" connection between NumStops and Stop). However, if the optional node has never been seen, we cannot test this relation. But based on Definition 13, the failure to predict a relation to an element that has never been seen is not a mistake (because it is not refuted by the current instance). We now make the following adaptations to *SimpleTaskLearn*:

– When an optional node first appears in a trace, add in all edges to and from this node not refuted by current instances. This is the same initialization that formerly happened only with τ_0.
– When edge semantics are tested, we still test every edge, even if one end contains no instances. The underlying semantics of m_λ determine the validity (count([], 0) can be true, but max([], 7) will come back false).

Although nodes and edges may now be initialized in episodes other than the first, the number of potential edges in the task graph as a whole is still $O(|S|(|G_I|+|G_O|)^2)$, so the resulting mistake bound for TGLA is on the same order as before (extra mistakes are made in adjusting the annotations of nodes but this quantity is dominated by the edge adjustments).

4.3. Selection Relations

We now relax assumption from Section 3 , which required that any time more than one part of a "whole" structure in the task graph repeated later in the task, the entire structure reappears. This assumption does not generally hold for common web service tasks. For instance, when buying a product on Amazon, after finding an offer, the buyer may need to enter the product and seller ids in a form, but should not have to copy in the manufacturer name, country of origin, or all the other non-essential properties of the product in this purchasing form. To model such "selection" of parts of a compound object, we introduce a new set of labels for semantic edges based on templates defined below, but first we cover a more concrete example of this situation.

In Figure 2, one of the flights from the Flight$^+$ list in ***FlightLookup*** appears exactly copied in the ***BookFlight*** output. Because of this, semantic links emanate from the parts of the Flight$^+$ node *and* the Flight$^+$ node itself (note the "member" link between Flight$^+$ and Flight). From these links, a reasoner could determine that the minimum priced flight should be chosen and that Id should be fed as input to ***BookFlight***. However, in general (and as we have seen in our Amazon.com and Google experiments), web services do not repeat the same exact structure between services. More commonly, a few elements of the larger structure appear after a member is selected from a list, as seen in Figure 3, where the "NumStops" node is omitted in the ***BookFlight2*** output. This omission prevents not only reasoning about how this flight was selected, but also modeling the connection between all the parts in the output of ***BookFlight2***. They are not just a random collection of the members of sublists from ***FlightLookup2***, they have a semantic connection based on a shared "whole" instance. These are stops for a single (and specific) flight. For instance, if the third flight is chosen, then the ***BookFlight2*** output should certainly not have a StopId of *STL*, but just having the "member" link does not assert this relation.

To address this issue, we introduce *derived* semantic relations based on two templates defined below. Labeled edges with these semantics will give us a way of predicting user preferences in selection as well as maintaining groupings of instances based on shared ancestry in the XML document. Our experiments with tasks comprised of services from Amazon and Google (see Section 6) indicate modeling such groupings are crucial for capturing the true semantics of tasks. Such preferences could be arbitrarily complex, so we focus here on a restricted subset of queries conforming to simple XPath (http://www.w3.org/TR/xpath) expressions of the form node[simplePredicate]/node/node/... . We do so as a proof of concept, to show how the sample complexity changes with these expanded semantics. Learning about more complex preferences is beyond the scope of this work.

The $Selection[n_{anc}, m', n_{pref}](n_1, n_2)$ template encodes a set of binary relations between two nodes n_1 and n_2, (like the two Eatery$^+$ nodes in Figure 3) where n_2 contains a subset of n_1's instances, and those instances were all parts of a larger "whole" structure above n_1 (e.g., eatery lists for all the stops on the same flight). Each relation instantiates the following:

– n_{anc} the "ancestor" of n_1 where the "whole" instance was chosen. (from $n_1 =$ Eatery$^+$ that's either Stop*, or Flight$^+$.)
– m', a semantic relation comparing two objects. For our purposes we assume that $m' \in \mathcal{M}' \bigcup \top$, where $\mathcal{M}' \subseteq \mathcal{M}$ and $\top$ is a wild-card relation explained below.

– n_{pref} is a child node in the same service graph (G_I or G_O) as n_1 and is reachable from n_{anc} by "part" edges without traversing another list node.

In Algorithm 1, each potential match for these nodes would have to be considered for the initial trace τ_0 (line 7) and all instantiations of these parameters in a *Selection* edge in G_T would have to be considered in the loop over current edges (line 16). Intuitively, *Selection* edges say that $\mathcal{I}(n_2)$ contains a subset of the instances in $\mathcal{I}(n_1)$ all of which descend from a single instance of n_{anc}, picked over others in that node because the corresponding instance at $n_{\text{pref}} = m'(n_{\text{pref}})$ (select an element of $\mathcal{I}(n_1)$ based on the value n_{pref}, tied to n_1 through n_{anc}). For instance, the Eateries in the ***BookFlight2*** service (n_2) are a subset of those in the Eatery$^+$ node for ***FlightLookup2*** (n_1), and were chosen from a flight (n_{anc}) based on min (m') NumStops (n_{pref}).

In our examples, we consider $\mathcal{M}' = \{min, max\}$. Because these functions will not be able to model all preferences, the wild-card $\top$ is used to indicate that *some* selection of an instance from a node is being done, but we cannot qualify it with the relations in $\mathcal{M}'$. This may introduce a number of redundant *Selection* relations (as can be seen with *Flight*$^+$ in Figure 3), but since all relations between nodes need to be valid when executing the tasks, these more general links not harmful. Overall, the number of possible instantiations of this template is $O(|\mathcal{M}'||S|d\max(|G_O|, |G_I|))$.

If two nodes in the same service graph (like Id and Eatery$^+$ in ***BookFlight2***) have Selection links with $m' \neq \top$, then their shared ancestry is easily checked from n_{anc} and n_{pref}. But, if $m' = \top$, one can't tell simply from the Selection links if, for instance, Id and StopId are chosen from the *same* flight instance. To combat this, we introduce a semantic relation that encodes such shared ancestry: $SameSelection[n_{anc}](n_1, n_2)$. We only consider this relation between nodes n_1 and n_2 in the same service graph (G_I or G_O) where both have Selection links referencing the same node n_{anc}. Considering both templates, we have expanded Λ from its original $(|\mathcal{M}|+2)$ to $O(|\mathcal{M}| + |\mathcal{M}'||S|d\max(|G_O|, |G_I|))$, leading to the following result.

Proposition 2. *Modifying TGLA (Algorithm 1) with the extended semantics described above makes* $O((((|G_O| + |G_I|)|S|)^2(|\mathcal{M}||S|d\max(|G_O|, |G_I|))$ *mistakes in the web-service task learning problem when the target semantics are representable with a Task Graph.*

Proof. [9] The full task graph has $O(((|G_O| + |G_I|) * |S|)^2$ nodes. Selection and SameSelection introduce $O(\mathcal{M}'|S|d\max(|G_O|, |G_I|)) + O(|S|\max(|G_I|, |G_O|)) = O(\mathcal{M}|S|d\max(|G_O|, |G_I|))$ edge labels. Each of these edges can be checked by simply grouping the instances as per the edge parameters (a polynomial time operation). Thus, the edges can be introduced as each node appears and checked for validity against each trace just as before. Finally, we recall that $d = O(\max(|G_I|, |G_O|))$, so the sample complexity is polynomial in the parameters of the task-learning problem. □

5. Reasoning with Task Graphs: Choosing Service Inputs

While the bulk of this paper concerns the problem of *learning* a Task Graph model from traces of users performing a task, we now analyze the problem where the agent has such a model and must select the correct inputs for each service based on the objects already encountered in the task. Note this problem of choosing the inputs for each service in the task is also a requirement of the online learning problem defined earlier (Definition 13), but here we turn our attention to the complexity of this reasoning process itself *given* a Task Graph.

We begin by formally defining the *Input-Selection* problem. The *Input-Selection* problem involves an agent performing a known task $T = \langle$S,R$\rangle$ comprised of services S= $[s_1...s_m]$ (see Definitions 11 and 9). Using the true Task Graph $G_T = \{N, E, \Lambda_{\mathcal{M}}\}$ and an initial SS-Graph Instance (e.g. the "Person" structure and objects in Figure 2), the agent must for each service s_i, provide an SS-Graph Instance that is a valid instance of G_{Ii}, the input graph for s_i. Validity refers here not only to the structural and semantic edges within G_{Ii}, but since this is a subgraph in the larger G_T, the objects in the instance of G_{Ii} must be valid for some instantiation of G_T, given that the objects for nodes corresponding to prior services ($s_j \prec s_i$) are already set. For instance, when picking an input to the **BookFlight** service in Figure 2, the one-node G_I does not encode any constraints, but the larger G_T (even though part of it would not yet be instantiated)

[9]The full proof of this Proposition is similar in form to that of Proposition 1, so here we simply outline the differences between the two

encodes that this input must be a member of the previous service's "id" list, and must correspond to the id of the lowest cost flight. Note that the latter reasoning requires looking *forward* to a portion of G_T that will not be instantiated until after this service call, but nevertheless encodes important constraints.

The service is then called with the objects in that instance. After each service call, the agent receives the output of the service (G_{Oi}), and can therefore instantiate that part of G_T (bind objects to nodes), so intuitively the problem is one of deciding the inputs to each service in a task, given a partially instantiated task-graph.

We note that this is not a learning task, so the worst-case bounds will be in terms of computation time, not sample complexity. The complexity of the input-selection problem is highly dependent on the mathematical relations $\mathcal{M}$ behind the set of edge labels in the given task graph. Below, we show how two different $\mathcal{M}$'s can change the input selection problem from trivial to intractable.

We begin with the simplest semantics, where $\mathcal{M} = \{=\}$, that is only equality relations between objects are considered. This corresponds to a task graph where all the semantic edges enforce equality between the instances corresponding to each node. This leads to the following simple result:

Remark 1. *With $\mathcal{M} = \{=\}$, the input-selection problem for a given service s_i with corresponding input graph G_{Ii} comprised of n nodes with maximum degree δ can be solved in $O(n\delta)$ time.*

The result is straightforward– for every node n_j in G_{Ii}, one can simply check every semantic edge and see if the connected node in G_T is instantiated and copy this value into n_j. If the connected node is not instantiated, then another link is checked. If no links lead to instantiated nodes, then an arbitrary object can be used because there are no constraints on previous services. Following longer paths of equality links is not necessary because of the transitivity of equality– any longer paths that lead to an instantiated node n_k means there will also be an edge between n_j and n_k because of our "maximal task graph" assumption [10].

However, as we have seen, task semantics usually require far stronger relations than equality, so we now consider a more expressive set of relations, $\mathcal{M} = \{member, sum, min, count, =\}$ where *member* can be used to mean that each element of a sublist is a member of a different list or sublist (see the explanation of the semantics in Section 4.1).

Remark 2. *The input-selection problem with $\mathcal{M} = \{member, sum, min, count, =\}$ in is NP-Hard.*

Proof. The reduction is from the well-known (and NP-Complete) Knapsack problem, where, given a number of items, each with a value v_i and cost w_i, one must determine if there is a collection of these items with value greater than or equal to V^* that does not exceed a total cost W^*. A task graph that encodes this exact problem (and could be constructed from an arbitrary instance of the knapsack problem) is shown in Figure 4. The member links from the instantiated output nodes to the input of the service encode the selection of items while the min, count, and sum relations in the output of the Knapsack service encode the value and cost constraints. The chosen items are stored in a sublist to facilitate the mapping of the member relation (each element in the sublist must be a member of the original list). The CostComp and ValComp nodes end up each containing lists of length two for comparing the packed weight and max weight and the packed value and minimum value (thus the member links between these comparison nodes and their elements as well as the min restrictions). A slightly simpler reduction is possible by introducing a *GreaterThan* relation and eliminating these comparison nodes. □

An exhaustive study of the complexity of inference in relevant classes of $\mathcal{M}$ is beyond the scope of this work, which is instead focussed on the sample complexity of learning. We provide the preceding results only to give examples in the general landscape and show that the inference required to use a learned task graph to perform a task may require super-polynomial computation, depending on the complexity of the semantics. While exact solutions to these problems are intractable, people using web services will often find approximate solutions meeting their constraints, including solutions that leverage meta-data or other hints that our system currently ignores.

6. Examples with Real Services

We now discuss several experiments where TGLA was applied to tasks comprised of publicly available

[10] In this case this assumption can actually be relaxed because even with a minimum number of equality edges the maximal graph can be reconstructed in (amortized) constant time using a unification-style algorithm.

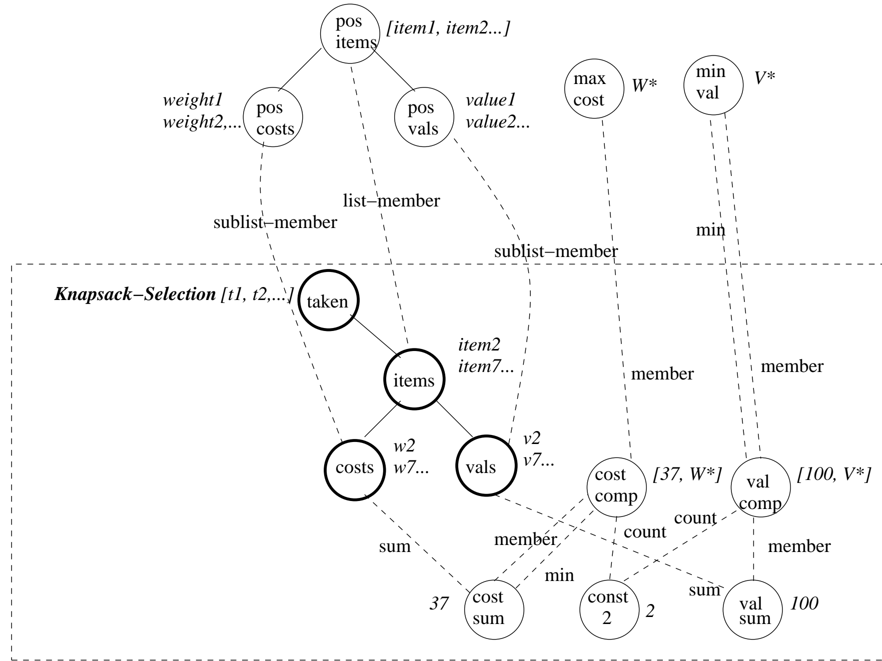

Fig. 4. A task graph encoding the knapsack problem. The CostComp and ValComp nodes end up each containing lists of length two for comparing the packed weight and max weight and the packed value and minimum value. Other constructions are possible using relations like GreaterThan, but here we show that even with the simple relations used in our experiments and examples, input-selection can be intractable.

web services based on traces collected by the authors. We begin with tasks where the services are all from the same provider. We then discuss a task where services from Google and Amazon are combined and show the system is able to learn its own homogeneous task graph despite the heterogeneous origins of the services. Summary results from the experiments, including the maximum number of traces needed, are reported in Table 6. The accompanying graph charts the number of services where mistakes occurred for seven episodes in the two most complex tasks we studied, averaged over 100 random orderings of the collected traces. While the total number of trace requests was generally the same in these runs, the quick descent of the curves indicates many instances of both tasks and many service calls within these instances can be completed correctly before all the nuances are learned. The full collection of traces are available at (www.research.rutgers.edu/∼thomaswa/traces.tar.gz).

6.1. Examples with Single Providers

Amazon.com offers an extensive web-service library (http://aws.amazon.com/aws), providing access to its inventory, customer wish-lists, and shopping carts. These services have been studied in the Web Service Composition (planning) community, where plans involving multiple Amazon services were dynamically constructed [15]. However, this automation relied on hand-crafted descriptions of each service. With an eye towards using our learned descriptions instead, we now present some results in the AWS testbed.

In the "AlbumBuy" experiment, an agent was given an artist and album title (tagged as "In1" and "In2", respectively) as well as a search index ("music") and a quantity to buy. The agent then had to find the corresponding ASIN (Item Id) and use it to create a shopping cart with the required number of copies of that item in it. The first trace provided to our agent involved a self-titled album (Title="Warren Zevon", Album="Warren Zevon"). This ambiguity resulted in the agent linking both In1 and In2 to both the "Title" and "Artist" inputs of ItemSearch. In the next episode (In1="Beatles", In2="Abbey Road"), the agent chose to send these parameters to ItemSearch backwards (e.g., "Beatles" to Album), garnering no results. The agent then received a trace and deduced the correct identity links. Other semantic relations mined in this task included (1) When ItemSearch returned multiple items (it matches substrings on titles), the one with the title matching In2 (and Title) should be added to the cart and (2) the quantity passed to CreateCart

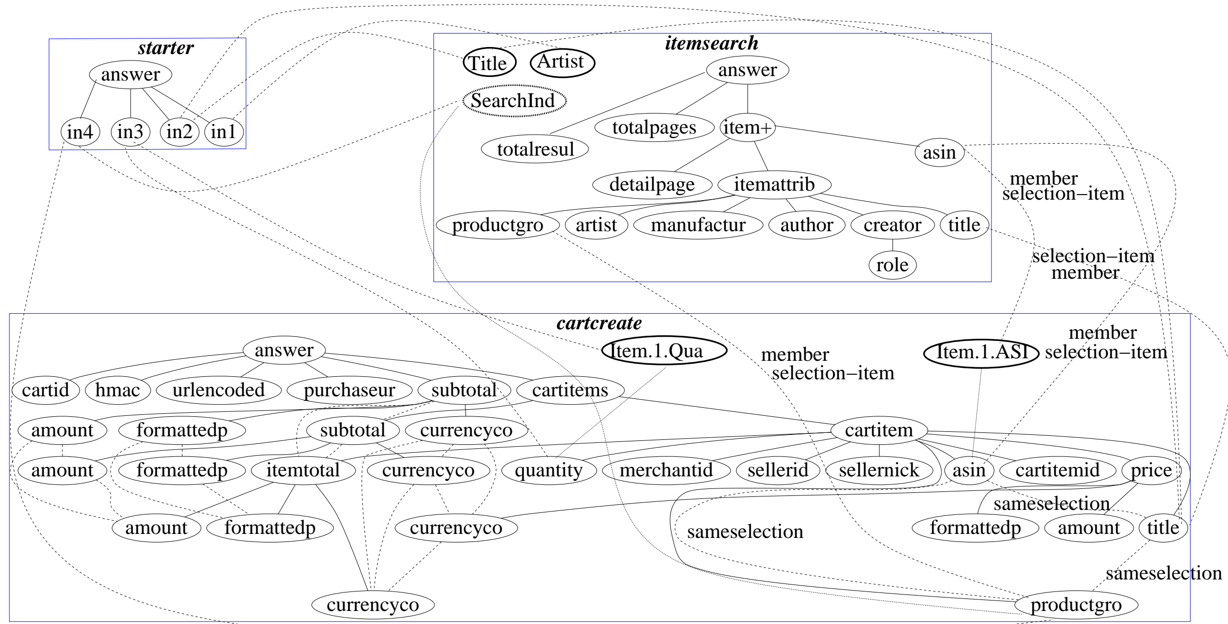

Fig. 5. G_T for the Amazon AlbumBuy task. Some relations are suppressed for readability.

should be linked to the requested quantity in the initial information– a non-trivial relationship since most of the traces requested only one album copy and many lists returned had a length of 1. [11] The former behavior was learned with the help of *SameSelection* relations, which linked the Title and ASIN of the item put in the cart. Since the Title was also linked back to the original input, the agent could infer which of the returned items to actually buy (by choosing the ASIN grouped with the matching title). Note that these behaviors are exactly right and the task would not be correctly executed without this knowledge. The full G_T for this task is illustrated in Figure 5.

We also experimented with tasks where agents learned about services for looking through wish-lists, searching for items (sometimes from earlier wish list searches), creating carts, and adding more items to a cart. Often the agents inferred rules for choosing items, such as buying minimum price items. Since exact structures are rarely copied, these rules were represented with the *Selection* and *SameSelection* relations. The mined graphs usually contain dozens of nodes, and typically fewer than 5 traces are needed before the learner can correctly execute tasks on arbitrary inputs.

In addition to the Amazon experiments, we also trained the system on tasks involving services from the Google Data API (http://code.google.com/apis/gdata/). Services in this library allow users programmatic access and editing capabilities to their email, contacts, spreadsheets, calendars, and other content. In this setting, we constructed a task for users filling out a form to receive reimbursements for travel on a per diem basis. Specifically, the traces tracked users looking up a conference, stored either as a (potentially multi-day) appointment or a series of appointments, in their Google Calendar. When multiple appointments were used to encode a multi-day trip, the system identified the min/max dates (selected from a sublist) as the beginning/end of the conference. The “where” field for the appointment was then used to look up the per diem information from a Google Spreadsheet fashioned from a real US government per diem spreadsheet, and then a form was updated with the traveler’s name, dates, and per diem information. This task involved several intricate relationships that needed to be learned. For instance, some of the per diem rates were listed seasonally (e.g. searching for the rate in Las Vegas returns a list of rates and dates, so the system had to pick the correct rate corresponding to the trip dates). With

[11] In other tasks, such links can be helpful. For instance, we ran another experiment where the quantity was not specified but traces showed it to always be one and the agent correctly learned this information was based on the “Count” of the ASINs passed to CreateCart.

Experiment	Nodes	Services	Max Traces Needed
Flight Booking	25	4	3
Amazon Album Buy	56	3	3
Google Per-Diem	147	4	5
Birthday Gift	231	6	7

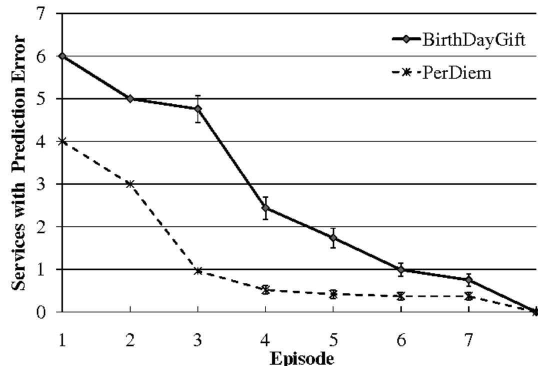

Fig. 6. *Left*: Sample results for learning task graphs from several examples discussed in this paper. *Right*: Number of services where mistakes were made (averaged over 100 orderings of task instances). A trace is not necessarily given after each episode, only when the learning agent makes a mistake, hence the 5 traces needed for Google Per-Diem are spread across 7 episodes.

our selection templates and adding *DateBefore* and *DateAfter* relations to $\mathcal{M}$, we were able to capture this behavior. Table 6 reports the maximum number of traces needed in our experiments and the number of nodes in the graph. The accompanying graph charts the number of services where prediction mistakes were made per episode, averaged over 100 orderings of the task instances. Note many of the task instances can be executed without error even before all the nuances of the task are learned, and often only one additional trace is needed after the third episode.

We also trained the system on similar tasks where different information had to be entered in the final form. For instance, we experimented using traces where appointments were only encoded using multiple single-day appointments (rather than the mixed approach above) and the length of the trip had to be entered in the final form, which the system correctly determined to be the count of the number of calendar entries. These examples demonstrate the flexibility of our approach— by learning task-dependent semantics of the services, it can adapt to slightly different uses of the services.

6.2. *Services with Different Providers*

One of the goals of the service-oriented computing movement is to compose tasks with service calls from different providers. Unfortunately, non-uniformity in web-service descriptions has made this goal quite difficult from a semantic perspective. While there has been work on learning unified service descriptions from heterogeneous providers in the ontology matching [12], semantic annotation [5], and even the machine-learning [10] communities, ontology matching requires semantic service descriptions (which are often not available, for instance Google and Amazon do not provide these), and the annotation techniques require an existing domain ontology (uncommon) and often try to mine universal descriptions from meta-data (web forms, WSDL files, etc.), rather than instances. In contrast, we have taken a more traditional machine-learning approach: because we are learning our own semantics based on traces, there is no need to reconcile mismatched or even missing descriptions.

To illustrate this point, we performed an experiment using services from both Google and Amazon. The task involved using the Google Calendar service to find all the birthdays of a user's colleagues within a given date range, then using the Google Contacts service to look up the email address of the person with the earliest birthday. This email address was then sent to the Amazon ListSearch service to find the user's Wish List. Then, the Amazon ItemLookup and CartCreate services were used to purchase the cheapest item on that list. Multiple traces were needed to learn certain finer points of this behavior, such as picking the earliest birthday, buying the cheapest gift, and eliminating erroneous date relations (such as relations based on publication dates). The system's success in inferring the semantic links in the task, even between multiple providers, shows that heterogeneity is not as vexing when learning from data, rather than analyzing sparse and potentially scarce description files. Also, because the Google Contacts service does not yet support full text indexing (instead returning a list of contacts), the selection templates were necessary for learning the correct behavior. Table 6 and the accompanying graph illustrate the number of services where mistakes are made on each episode. Notice that this task is more

complex than the earlier *Per-diem* task, but only takes a few more traces to learn, with half the services usually learned sufficiently for the experimental task instances after 3 traces. Also, many runs required fewer than the maximum (7) number of traces as more informative traces were encountered earlier. Given the complexity of the task, the small number of traces needed is a strong justification of this apprenticeship learning framework and our learning algorithm.

7. Extensions

We now describe two extensions of this work designed to make it applicable and usable in domains that might not fit all of the assumptions (specifically determinism or stationarity) made in the studies above, and discuss how to translate our learned models into more familiar planning operators. We begin by discussing services where relations in T^* may only hold with some probability and show that our data structures and learning algorithm can be adapted to such cases. We then discuss a variant of our core algorithm for the non-stationary case, and finally we describe how classical planning operators can be mined from task graphs.

7.1. Stochastic Services and Probabilistic Relations

Thus far, we have considered all services to be deterministic. That is, for a given input graph G_I, a service always produces the same output graph G_O. While the agent may not be able to predict the full instantiation of G_O (the object names themselves), the deterministic assumption did allow the algorithm to predict the semantic edges (and hence the relations between objects) expected in G_O. But suppose that a semantic edge e is only valid in 95% of instances of G_O. Under the deterministic assumption, once any of the 5% of outputs that do not contain e is encountered, Algorithm 1 will prune this edge and never bring it back. While the algorithm has made the correct choice for the deterministic case (the edge e is not guaranteed to hold in every instance of the task) this result is somewhat unsatisfying because e is valid in the vast majority of cases.

To analyze such a situation, we will make the simplifying assumption that each semantic edge e_i occurs *independently* with a probability $P(e_i) \in [0,1]$ and that there is no distribution to be learned on the appearance of missing elements, but as with our previous analysis, edge validity can still be checked when a node is empty (count([])=0). A more complicated analysis may be possible when we consider a distribution over missing elements.

Analyzing such situations is not possible in the original mistake-bound paradigm, which is only defined for deterministic hypothesis classes. However, the recent introduction of the *Mistake Bound Predictor* (MBP) framework [28] established a similar analysis tool with important properties for apprenticeship learning in the stochastic setting. Briefly, an MBP learner is one that makes a prediction on the probabilities of all the outcomes of a given action (in this case the relations that will hold after a service call). A *mistake* is made in this framework if the probabilities of these outcomes are outside of an ϵ tolerance (based on some norm) of the true probability distribution. An MBP bound is obtained by showing that, with probability $1-\delta$, only a polynomial number of such mistakes will be made for a given hypothesis class. We can now state the following about the learnability of G_T in the MBP framework under with the assumptions listed above.

Proposition 3. *The task graph representation with associated independent probabilities for each edge can be learned with an MBP bound of* $O(\frac{|\mathcal{M}|^2|G_T|^4}{\epsilon^2} ln(\frac{|\mathcal{M}||G_T|^2}{\delta}))$ *where* ϵ *bounds the sum of the errors over all edge probabilities, and where* $|G_T|$ *is the number of nodes in the true task graph.*

Proof. First, in the case where there are no missing elements and the edge probabilities are all independent, each service in a trace gives us a binomial sample for each edge's occurrence. Applying Hoeffding's inequality, we can state that each $P(e_i)$ can be accurately estimated (with probability $1-\delta$) after $\frac{1}{2\epsilon^2} ln(\frac{2}{\delta})$ samples. However, since we require the total error in probabilities to be $\leq \epsilon$, we need to ask for each $P(e_i)$ to be learned with accuracy $\frac{\epsilon}{|\mathcal{M}||G_T|^2}$. We also need to apply a Union Bound over all the edges (forcing their individual failure probabilities to be $\frac{\delta}{|\mathcal{M}||G_T|^2}$) to ensure a total failure probability of δ. Thus, after $\frac{|\mathcal{M}|^2|G_T|^4}{2\epsilon^2} ln(\frac{2|\mathcal{M}||G_T|^2}{\delta})$ samples, the joint probability of all edges will (with probability $1-\delta$) be learned with a total of ϵ error throughout the whole task graph.

When nodes may be optional (empty instances), the validity of edges can be checked even when nodes they connect to have no instances (count([])= 0). However, to check this validity our algorithm must at least know such nodes exist. Therefore, we will potentially need at worst $|G_T|$ mistakes/traces to reveal each node. But,

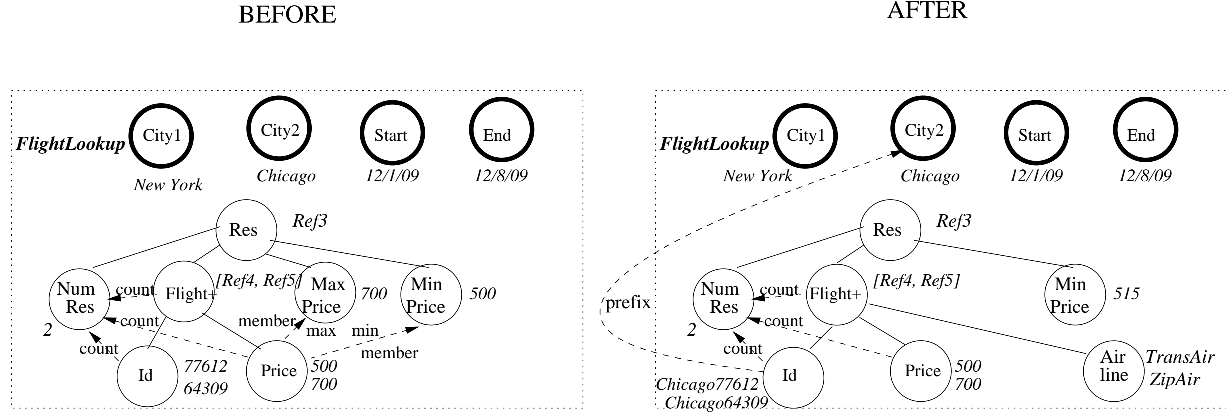

Fig. 7. The ***FlightLookup*** service from our earlier examples before and after a change has been made to the service itself. The changes are the introduction of the Airline node, the elimination of the MaxPrice node, the change in the semantics of the MinPrice node (no longer just the minimum member of price as it now includes taxes), and a change in the Id's that makes the prefix relation true.

we can simply add in these extra mistakes and the MBP bound remains $O(\frac{|\mathcal{M}|^2|G_T|^4}{\epsilon^2}ln(\frac{|\mathcal{M}||G_T|^2}{\delta}))$. □

This analysis shows that under simplistic conditions (where each edge's validity occurs independently), task graphs with probabilistic edges are efficiently learnable. In cases whee these independence assumptions do not hold, learning algorithms should be able to leverage assumptions such as edges being conditionally independent of other edges given their neighboring to avoid learning about the full joint probability of edge occurrences, but we leave such analysis to future work.

7.2. Non-stationarity: Adapting to Changing Services

While large-scale changes in service behavior are usually announced by the service providers, small changes, such as the addition of content (like changing the format of a date or the name of a publisher of a book now being reported) are often made without much fanfare. Anecdotally, we saw these sort of changes even during our own study of Google web services, where minor changes in the structure or content of the XML responses caused changes to the task graph that would not have happened without the format shift. However, our data-driven approach can be extended to handle such small variations, and automatically refine its task-graph model. We now outline and briefly demonstrate how such a procedure would operate.

As an example of a changing service, consider the "before and after" depiction of a slightly modified flight lookup service in Figure 7. The original version (left) is the same as the ***FlightLookup*** from Figures 1 and 2, but several changes are made in the newer version (right). First, there is a new node indicating the Airline running each flight, and the MaxPrice node has been eliminated. Also, the MinPrice node is now meant to represent the minimum *total* price (with fees and taxes), and therefore the "min" and member relations between this node and the Price node no longer hold. Finally, the instances of the Id node contain the name of the destination city prepended onto the original Id string. To capture this, we can consider an extra relation in $\mathcal{M}$ called "prefix", which is true when an instance is a prefix (but not the equivalent) of another. Notice that in the original ***FlightLookup***, all such relations would have been eliminated during learning.

We now consider the problem of noticing such changes and adjusting our task graph accordingly. We note that this problem is one of determining non-stationarity, a particular vexing problem for most machine learning algorithms and the inherent difficulties of such problems appear here as well. For instance, when seeing a node added for the first time, does that indicate that a shift has been made in the service description or that the node was just optional and hitherto not seen? Without assumptions on the probability of optional nodes, such questions will not be generally answerable, so we assume that every optional node will appear with probability at least ρ while a service has not changed.

In the task graph representation, there are four kinds of service changes that may be encountered, each of which occurs in the example from Figure 7. The first two are that a node may be added (Airline) or deleted

(MaxPrice). Such changes can be handled by the current algorithm by annotating the node as optional (similar to the adjustments from Section 4.2), which initially is the correct thing to do. But as mentioned above, if the service really has changed, then this node is not really optional, so if a parameter ρ is known, we can later discard this annotation for an added node. Similarly, for a deleted node like MaxPrice, we can initially annotate it as optional when it first disappears, and then once the probability of the node slips under ρ, discard the node entirely. Thus we can change the adapted block from line 22 of Algorithm 1 (which has already been adapted to handle $*$ and $?$ notations in Section 4.2) to perform the following operations:

- Let w be a sliding window that is no larger than the total number of traces seen so far. On every episode, update p_i to be the frequency of node n_i appearing in the last w episodes.
- If a node n_i has just appeared (thus receiving an annotation of $*$ or $?$), set $p_i = \frac{1}{w}$.
- If a node n_i has its annotation changed to $*$ or $?$, set $p_i = \frac{w-1}{w}$.
- If a node n_i with a $*$ or $?$ annotation has $p_i = 1$ then remove the annotation.
- If $p_i < \rho$, delete n_i and all edges leading in or out from it.

In our example, these rules cause the system to initially add Airline as an optional node ($?$ annotation), but after enough examples, this annotation will be removed. Also, the MaxPrice node will be initially marked as optional, but eventually eliminated as it does not reappear after the change. This takes care of nodes that changed when the service shifted, and since all possible edges are considered when new nodes are introduced, these rules will also cover new edges for the new nodes.

Now we consider the case where the same nodes exist, but semantic edges have been added or deleted due to the service change (the prefix edge and the deleted min and member edges to MinPrice in the updated ***FlightLookup***). In the latter case, our deterministic algorithm will simply eliminate edges that do not appear in a trace after the change. However, to "bring back" edges that were previously eliminated (like the prefix edge), we must recheck every edge that appears in a trace to see if it has reliably appeared in the recent window w. We can do so by keeping track of the probability of each edge over a sliding window w and only reporting edges that occur more frequently than ρ.

Using the modifications described above, our learning system can successfully adapt and even alert users to small changes in a service description. Alerts can be triggered whenever edges are added back into G_T or whenever optional annotations are removed or nodes are deleted, all of which signal a definitive change in the service definition. However, we note that the monotonic convergence of our learning algorithm (and the accompanying mistake bounds) are lost in the nonstationary case, but this is to be expected since the target task is shifting over time.

7.3. Mining Planning Operators

A number of planners have been developed in the web-service composition community [6,14,15]. While this paper focussed on learning task-dependent semantics in the form of a task graph G_T, we now discuss future work on an extension for mining task-independent operators (as in Table 1) from such graphs. The goal of such an effort would be to mine operator descriptions of individual services from the graph by treating the output nodes as literals added to the current *state* and the links to previous services as preconditions for executing the service. A number of planners have been developed in the web-service composition community [6,14,15] [12] that could be used with these mined operator descriptions since they generally cover richer languages than our own.

There is no standard language for web-service operators, though many proposals exist (OWL-S, WSDL-S, etc.). Almost all share the idea of input and output parameters for the service, and pre/post-conditions involving them. These common structures can be derived from a task graph by treating the output nodes as literals added to the current *state* and the links to previous services as preconditions for executing the service. However, the target language will have an impact on how much information from G_T is actually ported to the new operators. For instance, if the output nodes are linked to nodes from previous services that are not linked to the inputs, then new variables that will already be grounded at the service's invocation need to be introduced into the operator's *scope* (as with "Home" and "DestCity" in Table 1), reminiscent of the injection of *deictic* references [19]. But if a language

[12] Interestingly, the last planner listed considered planning with operators derived (by hand) from Amazon Web-Service descriptions, which served as our real world testbed, so this may be a natural vein for future work.

FlightLookup(City1, City2, Start, End)
PRE: *Equal*(City1, Home), *Equal*(City2, DestCity)...
ADD:*Result*(X), *NumRes*(Y), *Flight*(Z), *Part*(Z, X), *Part*(Y, X), *ListCount*(Y, Z)...

Table 1

Partial planning operator from the **FlightLookup** service from Figure 2

with more limited scope (similar to STRIPS+WS [27]) were used, these links would not be represented in the operator, potentially making the algorithm faster, but sacrificing richness in the semantics.

Since G_T is built from a single task, these descriptions will be heavily biased, but they could then be refined using other tasks. That is, several G_T's can be merged by dropping edges that do not appear in all the graphs. Links between nodes within a service (such as the "Count" relationship between NumRes and Flight$^+$) can easily be resolved in the unified version, but links to nodes outside of the service (e.g. preconditions based on the links back, as in the **FlightLookup** operator) are more complicated as they require a stronger language to maintain the same semantics as the task graph, for instance either a disjunction or more intimate knowledge of the types of each node. This is where the connection between input/output nodes and ontologies would be helpful, and hence the combination of this work with the earlier-mentioned work on learning of service parameter classification, or semi-automatic schema merging between the XML tags in examples and ontologies, as considered in [20] seems advisable. But notice that even without these inter-service links, service descriptions with limited semantic scope (similar to the scoping restrictions of STRIPS) can be achieved. Such scoping restrictions would again make the language more restrictive than the general task graphs we have presented, but the size of the operators would likely be more reasonable.

8. Related Work

A previous application [3] of Inductive Logic Programming (ILP) showed promise in learning web-service descriptions from examples, based on known descriptions of other services. Unlike their approach, which relied on heuristic search and "sufficient" data, we have focussed on algorithms that can guarantee high performance with a limited amount of data. Our earlier work [27], performed a sample complexity analysis of learning planning operator descriptions in a restricted language, but did not explicitly consider relations between operators, and the operators had extremely limited scope. The Task Graph representation itself bears resemblance to the structure from Simultaneous Learning and Filtering (SLAF) [23]. However, unlike SLAFs our task graphs cannot represent arbitrary boolean formulas, but do have positive sample efficiency results.

A separate track of research focusses on creating web-service descriptions for heterogeneous sources based on a central ontology. This thread includes acquiring descriptions using text-mining algorithms on the service's documentation [5], and the use of ontology merging techniques applied to full (but not directly compatible) semantic descriptions of services [12,7]. However, both require an existing domain ontology and large amounts of service documentation, and even in more adaptive variations that use service instances [11,10], the focus on universal descriptions and semantic annotation differs from our goal of mining task specific relations between concepts directly from relatively (compared to full semantic descriptions) easy to find collections of XML documents used to communicate to and from the services.

There has also been work on better interfaces for non-technical users to specify web service descriptions. This includes mashup interfaces for expert technical users [16], mechanisms for mashing up service user interface components [4], and interfaces for users with varying levels of technical skills, including a basic spreadsheet-style interface [18]. However, our work differs from these efforts because it does not require users to have any technical abilities beyond what they already use to complete their task, and our system can learn service and task descriptions that the user may not even be able to describe. This last point is critical as users in a complex workflow often describe "knowing how" to execute the process but being "unable to explain exactly how they do it". Also, when users are comfortable using interfaces as described above, and when they have a fair amount of background knowledge they wish to directly encode, these intuitive interfaces could be used to bootstrap a task-graph, and our apprenticeship learning system could then be used to fill in missing relations that seem to occur frequently.

Another area that future iterations of this work can draw upon is the field of workflow induction (also known as "process mining") [26]. This field is concerned with inducing models of tasks (but usually only the sequence of calls, not the dataflow) that contain

loops, conditions, and concurrency, usually representing the task with a Petri Net [17]. Learning such powerful structures is inherently intractable (Petri Nets can represent Context Free Grammars), and even restricting the form of these nets usually leads to super polynomial learning times [26]. However, with certain restrictions systems from the field of workflow induction [25] could be used to learn these more complex versions of S . Along the same lines, the overall problem of learning the structure and flow of web services bears resemblance to the problem of program induction [8], which has been studied empirically with heuristics, but has not produced sample complexity results.

More recent work on Workflow Induction from Traces (WIT) [29] has moved from just learning the sequence of calls in a workflow to also modeling the dataflow. These dual goals are more in line with our own, though in this paper we have focused almost exclusively on the latter. The WIT algorithm considers a larger class of workflows ("witty workflows") than our work in terms of the service calls (S) and employs powerful grammar induction algorithms to extract structures like loops and concurrency. However, when modeling the dataflow (R), WIT considers only equality (so $\mathcal{M} = \{=\}$), so in that sense they consider a more restricted language. WIT is both a sound and complete algorithm for learning the more complicated structure (as well as the more restricted dataflow). Our work has made harsher restrictions to the shape of the workflow to ensure that only a small number of traces (rather than just a countable amount) are needed to learn the task. However, we have considered a far more expressive language for modeling the dataflow. Thus, there seems to be a promising future in combining these two algorithms using the grammar induction (and some heuristics) from WIT to learn complex service call patterns, with TGLA used to learn a semantically rich dataflow. Future work combining these two properties could be quite fruitful, especially for modeling error handling and responses to transaction failures. In such cases, a process mining component could learn to identify branches of the control flow that lead to failure and methods for recovering from it, but the dataflow learner (our system) would be responsible for learning *why* such failures were occurring (based on the learned semantic relations). In this way, our system and representation provides a potentially useful complementary algorithm for process mining methods.

Finally, a number of planners have been developed in the web-service composition community [6, 14,15] [13]. These works considered the problem of determining what sequence of service calls to make to accomplish a goal when *given* models of how the services work (relations between inputs and outputs). Our work can be seen as complementary to theirs because we are learning (task dependent) models from data, though Section 7.3 discussed interpretations of our Task Graph model that could be used with these planners.

9. Conclusions

This paper described and analyzed practical algorithms for building web-service task descriptions from traces of users completing these tasks. We showed that interesting syntactic and semantic structures could be learned from these traces, including learning about missing elements, and relations related to element selection, which are very common in web-service domains. We also emphasized empirically frequent data structures such as lists and operators over them, including aggregates (sum, avg, etc.), which are crucial to describing functional semantics of real services. A chief novelty of this paper is the theoretical results proving that the learning can be done efficiently– the number of traces required to learn the full task description grows polynomially with the size of the task and the richness of the semantics. Finally, we deployed this system in real-world testbeds involving tasks using Amazon and Google services, and showed that these tasks could be learned without a priori semantic information for these domains even when tasks involved services from multiple providers. This innovative application of machine-learning techniques and sample complexity analysis provides a unique perspective on developing web-service descriptions not beholden to pre-packaged or hand-tooled definitions.

Acknowledgements

We thank Google and Amazon for providing a large and diverse set of public web services that were instrumental to the experimental section and the development of the algorithms. This project is supported by DARPA IPTO under contract FA8650-06-C-7606.

[13] Interestingly, the last planner listed considered planning with operators derived (by hand) from Amazon Web-Service descriptions, which served as our real world testbed, so this may be a natural vein for future work.

Appendix

A. Appendix: Proof of Proposition 1

Here, we present the efficiency proof for the Simple Task Graph Learning Algorithm (Algorithm 1). The proofs for the more complex task graphs (for instance those using the Selection templates) follow the same form.

We begin with a lemma showing that learning the true task graph G_T^*, which is a task graph whose edges correspond exactly to the structural edges in the XML structure graphs of the services in S and semantic edges corresponding to the real semantic relations (R) of the task, is sufficient for guaranteeing no mistakes will be made.

Lemma 1. *Using the true task graph G_T^* to make predictions and choose inputs in the task learning problem will not produce any mistakes.*

Proof. We consider all the possible sources of a mistake. First, an agent could provide the wrong inputs to a service. But using G_T^*, the agent knows all the valid semantic links between the input to the service and the currently known objects, so the agent will be able to evaluate each potential input o' by checking each semantic edge $e = \langle n_1, n_{in}, \lambda_m \rangle$ leading into the input node n_{in} (by evaluating $m(\mathcal{I}(n_1), o')$. We note that for complex semantics the reasoning may require super-polynomial computation, as covered in our discussion of selecting inputs (Section 5). However, this does not affect the sample complexity.

Mistakes can also be made when the agent declares what semantic relations will hold between elements of the task, but since edges in G_T^* appear if and only if they correspond to relations in R , this cannot happen.

The only other way to make a mistake is to make an incorrect prediction about the structure of a service, (in this case predicting a $^+$ annotation) but the nodes of G_T^* have the true annotations of the true XML structure graph for all the services, so such a mistake cannot be made. □

Now we will show that Algorithm 1 converges to the true task graph G_T^* in the realizable setting with no more than a polynomial (in the relevant quantities) number of mistakes.

Proposition 4. *SimpleTaskLearn makes $O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ mistakes in the simple web-service task-learning problem when the target semantics are representable.*

Proof. The initial Task Graph (T_{G0}) is constructed from the task instance τ_0 and then refined in subsequent episodes. T_{G0} is made up of the XML structure Tree and semantic edges that are valid with respect to τ_0.

There are at most $((|G_O| + |G_I|)|S|))$ nodes in the entire task graph which means there are $((|G_O| + |G_I|)|S|)^2$ possible unlabeled edges, which is then multiplied by the number of potential labels, $|\mathcal{M}|$ (since $|\mathcal{M}| + 2 = |\Lambda|$) to give us $O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ possible semantic edges represented in τ_0.

In each episode, the current G_T is used (as in Lemma 1) to choose the inputs and make predictions about what relations hold and what structure will be seen in each subsequent service instance. If multiple edges leading into a node suggest different semantics (for instance if a "min" and "max" edge both appear between two nodes), one is picked arbitrarily.

For each of the predictions made based on semantic edges $\langle n_1, n_2, \lambda_m \rangle$, one of three cases applies (where we use $m(n_1, n_2)$ as shorthand for $m(\mathcal{I}(n_1), \mathcal{I}(n_2))$):

1. The prediction is correct, $m(n_1, n_2)$ is true, and no other edges $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ were proven invalid. No action needs to be taken.
2. The prediction is correct, $m(n_1, n_2)$ is true, but another edge $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ has been shown to be invalid with respect to the current task instance. The latter edge is removed from the graph.
3. The prediction is incorrect, $m(n_1, n_2)$ is false, resulting in a mistake. This edge, and all other edges $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ that are invalid with respect to the trace (task instance) are removed from the graph.

The worst case, in terms of the mistake bound, is that all the graph refinements after τ_0 occur because of case 3 and that only one refinement happens per task instance. However, because there are at most $O((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ edges in G_T to begin with, and because edges are never added back into the graph once they are removed, only $O((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ such mistakes can be made.

All that is left now is to bound the number of mistakes made when predicting the $^+$ annotation (other annotations later in the paper are dealt with similarly). There are $O(|G_O| + |G_I|)|S|)$ nodes, and initially all of those that are not lists in τ_0 are considered singletons (no annotations). The algorithm can only make mistakes when predicting these cannot be lists when

in fact they turn out to be lists, at which point their annotation is changed to $^+$ and never changes back. Since there is evidence that these elements can occur as lists, this must be the true annotation for the node in the service. This annotation learning introduces $O(|G_O| + |G_I|)|S|)$ mistakes.

In the realizable case, where G_T^* exists, the edges in T_{G0} but not G_T^* will be eliminated and all incorrect annotations will be made (a simple inductive argument shows that the same mistake is never made twice by the agent), so the algorithm converges to G_T^* and makes at most $O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}| + (|G_O| + |G_I|)|S|)) = O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ mistakes. □

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[2] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *Proceedings of the Thirty-Second International Conference on Very Large Data Bases*, 2006.

[3] Mark James Carman and Craig A. Knoblock. Learning semantic definitions of online information sources. *Journal of Artificial Intelligence Research*, 30:1–50, 2007.

[4] Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati, Chang Heng, and Li Yan. From people to services to ui: Distributed orchestration of user interfaces. In *Proceedings of Business Process Management - Eighth International Conference*, 2010.

[5] Andreas Hess and Nicholas Kushmerick. Learning to attach semantic metadata to web services. In *Proceedings of the International Semantic Web Conference*, 2003.

[6] Jörg Hoffmann, Piergiorgio Bertoli, and Marco Pistore. Web service composition as planning, revisited: In Between background theories and initial state uncertainty. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*, 2007.

[7] Jingshan Huang, Jiangbo Dang, and Michael N. Huhns. Ontology reconciliation for service-oriented computing. In *Proceedings of the IEEE International Conference on Services Computing*, 2006.

[8] Ryutaro Ichise, Daniel Shapiro, and Pat Langley. Structured program induction from behavioral traces. *Systems and Computers in Japan*, 36(11):730–740, 2005.

[9] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the Association for Computing Machinery*, 29(3):699–717, 1982.

[10] Matthias Klusch, Patrick Kapahnke, and Ingo Zinnikus. Sawsdl-mx2: A machine-learning approach for integrating semantic web service matchmaking variants. In *Proceedings of the International Conference on Web Services*, 2009.

[11] Kristina Lerman, Anon Plangrasopchok, and Craig A. Knoblock. Automatically labeling the inputs and outputs of web services. In *Proceedings of the Twenty-First Conference on Artificial Intelligence*, 2006.

[12] Qianhui Althea Liang and Herman Lam. Web service matching by ontology instance categorization. In *Proceedings of the IEEE International Conference on Services Computing*, 2008.

[13] Nick Littlestone. Learning quickly when irrelevant attributes abound. *Machine Learning*, 2:285–318, 1988.

[14] Zhen Liu, Anand Ranganathan, and Anton V. Riabov. A planning approach for message-oriented semantic web service composition. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*, 2007.

[15] Annapaola Marconi, Marco Pistore, Piero Poccianti, and Paolo Traverso. Automated web service composition at work: the Amazon/MPS case study. In *Proceedings of the International Conference on Web Services*, 2007.

[16] E. Michael Maximilien, Ajith Ranabahu, and Karthik Gomadam. An online platform for web apis and service mashups. *IEEE Internet Computing*, 12:32–43, September 2008.

[17] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[18] Željko Obrenović and Dragan Gašević. Mashing up oil and water: Combining heterogeneous services for diverse users. *IEEE Internet Computing*, 13:56–64, 2009.

[19] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[20] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proceedings of the Thirteenth International World Wide Web Conference*, 2004.

[21] Marco Pistore, Luca Spalazzi, and Paolo Traverso. A minimalist approach to semantic annotations for web processes compositions. In *Proceedings of the European Semantic Web Conference*, 2006.

[22] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[23] Dafna Shahaf. Logical filtering and learning in partially observable worlds. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

[24] L. G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

[25] W. van der Aalst and A. Weijters. Process mining: A research agenda. *Computers in Industry*, 53(3):231–244, 2004.

[26] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[27] Thomas J. Walsh and Michael L. Littman. Efficient learning of action schemas and web-service descriptions. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*, 2008.

[28] Thomas J. Walsh, Kaushik Subramanian, Michael L. Littman, and Carlos Diuk. Generalizing apprenticeship learning across hypothesis classes. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning (ICML)*, 2010.

[29] Fusun Yaman, Tim Oates, and Mark Burstein. A context driven approach for workflow mining. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence*, 2009.