Learning Information-Gathering Procedures by Combined Demonstration and Instruction

James Blythe
Dipsy Kapoor
Craig A. Knoblock
Kristina Lerman
USC Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Steven Minton
Fetch Technologies
2041 Rosecrans Ave, El Segundo, CA 90245

Abstract

Existing systems are able to learn information agents through demonstration that provide programmatic access to web-based information. However it is still difficult for end users to combine these information agents in procedures that are customized to their particular needs. We combine learning by demonstration with learning by instruction to build a system to learn such procedures with a small amount of human input. The instruction system relies on knowing the input-output types of the information agents in order to combine them. We make use a system that learns to predict the types from examples to simplify this part of the task. Our instruction system performs a search that has interesting similarities with proof search in explanation-based learning.

Intelligent software agents aim to assist users in the office environment by carrying out complex everyday tasks, for example planning travel, or managing the purchasing of equipment. These tasks are often on-going processes, where the assistant should initially combine information from a variety of heterogeneous sources and process the information as the user wishes, perhaps monitoring the progress of the task and continuing to give help where appropriate. For example, in planning a trip, the assistant may initially gather information about flights and hotels based on the user's preferences and budget, make some recommendations and assist with booking the user's choice, send reminders to the travelers, monitor for changes in prices, and check in for flights at the appropriate time. The software assistant must be able to learn new tasks and procedures, due to the wide range of potential tasks the assistant may be called to perform and the range of preferences individual users have about how to perform each task.

No single learning approach is appropriate across the range of tasks to be learned by the intelligent system. Our initial approach has combined learning by demonstration with subsequent learning by instruction, following the style most convenient for the user at each stage. For example, users can highlight the part of a web page containing target information such as a hotel price, but cannot typically provide an algorithm to retrieve this price, in part because the algorithm may rely on patterns in the page's html representation while the user follows visual and cognitive cues. For this we use programming by demonstration. Later the information is combined with other sources in a task to find appropriate hotels based on price and distance from a meeting point. It is typically more efficient for users to describe the criteria and associated actions rather than provide demonstrations, which might require several episodes in a simulated environment that has to be designed and maintained. For this we use programming by instruction. To be able to combine several tasks, each learning approach should accept procedures built using the other approach, and this requires them to be aligned to an ontology of inputs and outputs at least. We use a classifier learned from data to suggest to the user a short list of alignments with the ontology; the user can accept the default choice or switch to an alternative.

Our target language is SPARK (Morley & Myers 2004), a plan execution language that supports programmatic composition of substeps. Although SPARK procedures may have preconditions and effects, the system does not typically construct plans based on subgoal search. Procedures typically have fewer constraints than plans in most HTN planners. The existing procedures and their signatures, along with the concept ontology and the signatures of available queries on concepts, do provide enough structure to guide interpretation of a user's instructions to modify procedures, however. This process has similarities to explanation based learning, as we describe below. The technologies we have integrated are (1) EzBuilder, to create agents for programmatic access to semi-structured data on the web, such as online vendors or weather sites, (2) PrimTL, to map the input and output parameters used by these agents to concepts in a predetermined ontology and (3) Tailor, to create procedures that compose other procedures, EzBuilder agents and queries, with iteration and branching, based on user instructions in text.

In the next section we introduce a simple example problem, and briefly describe the three component tools used in our solution. We then summarize the results of tests with a small number of target procedures. Next we draw parallels between learning by instruction as performed by Tailor and explanation-based learning, in which the search for a code fragment that matches the user instruction mirrors the search for a proof that an example belongs to the target concept.

Example problem and component systems

Suppose the user is planning to travel and wishes to make use of a new online site for hotels and rates. First she builds a procedure to access the site, requiring a city name and returning a list of hotels, with a set of features available for each hotel. Next, she creates a procedure that uses the first one to find a set of available hotels within a fixed distance of a location. This procedure can be called with different locations, dates and distance thresholds. After choosing a hotel, the user would like to be informed if the rate subsequently drops. She creates a third procedure, also using the first one, that the assistant will invoke daily to check the rate against the booked rate, and email her if it has dropped.

EzBuilder: Automating access to web information sources

Fetch Technologies' **EzBuilder** tool helps the user create agents that can automatically query and extract data from information sources. Most online information sources are designed for humans, not computer programs, affecting how the site is organized and how information is laid out on its pages. The hotel reservations site shown on the right in Figure 1 allows user to search for hotels available in a specified city on specified dates. To build an agent for this site, the user demonstrates to EzBuilder how to obtain the required information by filling out forms and navigating the site to the detail pages, just as she would in a Web browser.

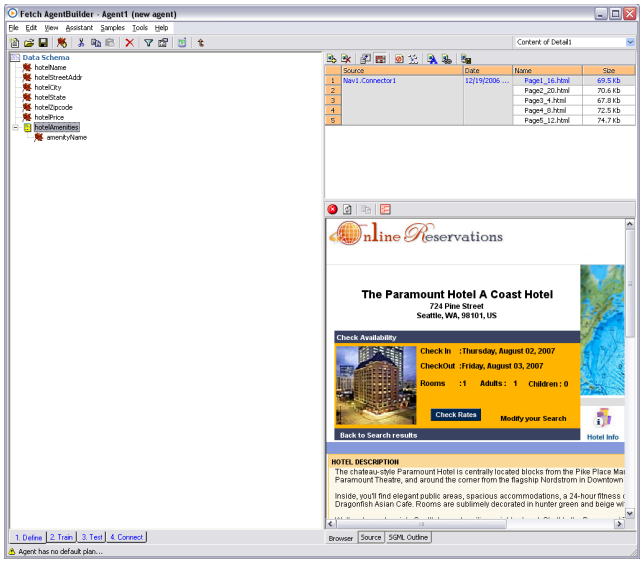

Figure 1: Schema definition for a hotel agent using the "Online Reservations" site. The user defines output fields such as "hotel name" and "hotel city", then drags the relevant text from the page over each output field for each page used in training.

As the user navigates the site, EzBuilder creates a model describing how to find pages of interest for a query. Next, the user specifies what data she wants from the site (in this case, name, address, price etc.) and trains EzBuilder to extract it, by dragging the relevant text on the page to a schema instance created by the user as shown in Figure 1. EzBuilder then analyzes the HTML and learns extraction rules for the required data (Muslea, Minton, & Knoblock 2001). Typically one to three samples are needed. However, if there is some variation in the format or layout of the page, the user may need to mark up additional samples so that the wrapper can learn general enough extraction rules.

PrimTL: Aligning primitive procedures to a type hierarchy

Although the newly created agent is now available for querying information sources, it cannot be used programmatically by other CALO components because its input and output parameters are not yet aligned with a common ontology. This is done by **PrimTL**, automatically launched after EzBuilder exits, which assists the user in semantically annotating the input and output parameters used by the source and linking them to a common CALO ontology.

Our classifier learns a model of data from known sources and uses the models to recognize new instances of the same semantic types on new sources (Lerman, Plangprasopchok, & Knoblock 2006). The classifier uses a domain-independent pattern language (Lerman, Minton, & Knoblock 2003) to represent the content of data as a sequence of tokens or token types. These can be specific tokens, such as '90292', as well as general token types, '5DIGIT' or 'NUMBER'. The general types have regular expression-like recognizers.

Data models can be efficiently learned from examples of a semantic type. We have accumulated a collection of learned models for about 80 semantic types using data from a large body of Web agents created by our group over the years. We can use the learned models to recognize new instances of the semantic type by evaluating how well the model describes the instances of the semantic type.

PrimTL reads data collected by EzBuilder, and uses the classifier to present to the user a ranked list of the top choices of the semantic type for each parameter. The semantic labeling step for the OnlineReservationZ agent is shown in Figure 2. The top scoring choice for each semantic type is displayed next to the parameter label. The user can see the other choices, and select one of them if appropriate.

Tailor: customized composition of information sources

Once the agent is aligned with the ontology, it can be used in the body of compound procedures that combine calls to other procedures along with iteration and branching. **Tailor** allows the user to create procedures by giving short instructions about the steps in the procedures, their parameters and conditions under which they are performed. By mapping the instructions into syntactically valid fragments of code and describing the results in a natural interface, Tailor allows users to create executable procedures without detailed knowledge of the syntax of the procedures or the ontology they use (Blythe 2005a; 2005b).

Tailor helps a user create or modify a procedure with an instruction in two main steps: *modification identification*

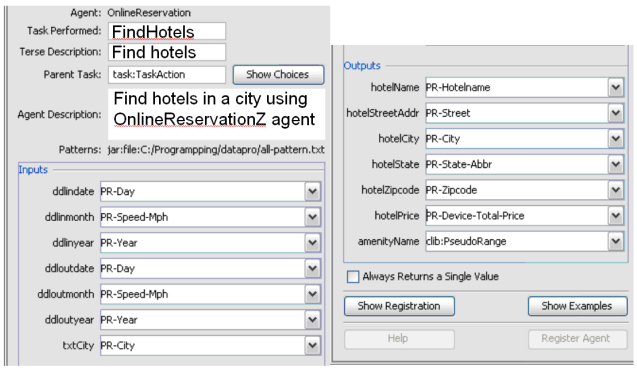

Figure 2: PrimTL's semantic mapping editor uses the training examples to predict the type of the agent's inputs and outputs based on their content. The user can correct a prediction if needed, choosing from PrimTL's top choices or from the whole type hierarchy.

and *analysis*. In *modification identification*, Tailor classifies a user instruction as one of a set of action types, for example adding a new substep, or adding a condition to a step. The instruction "only list a hotel if the distance is below two miles", for example, would be classified as the latter. This is done largely by keyword analysis.

Tailor also searchs for the step or condition parameters. For example, the phrase "if the distance is below two miles" will be used to provide the condition that is being added to a step. The mapped condition may require several database queries and several auxiliary procedure steps. In the current example, a separate procedure that finds distances based on two addresses from an online source is called before the numeric comparison in the condition, and provides input to it. Tailor finds potential multi-step and multi-query matches through a dynamic programming search (Blythe 2005a). The ability to match multiple actions and queries is essential to bridge the gap between the user's instruction and a working procedure, and is used in the interpretation of most instructions.

In *modification analysis*, Tailor checks the potential procedure change indicated by the instruction to see if any new problems might be introduced with the set of procedures that are combined within the new procedure definition. For example, deleting a step, or adding a condition to its execution, may remove a variable that is required by some later step in the procedure. If this is the case, Tailor warns the user and presents a menu of possible remedies, such as providing the information in an alternate way.

Tailor maintains the procedure internally in SPARK's format as it is being built, and shows the user an automatically-generated text description. A SPARK-like version of the procedure being developed in Figure 3 is as follows.

```
(defprocedurelistHotelsNearAtProc
  doc:"AddedbyTailor"
  cue:[do:(listHotelsNearAt?meeting?location)]
  precondition:(True)
  body:
```

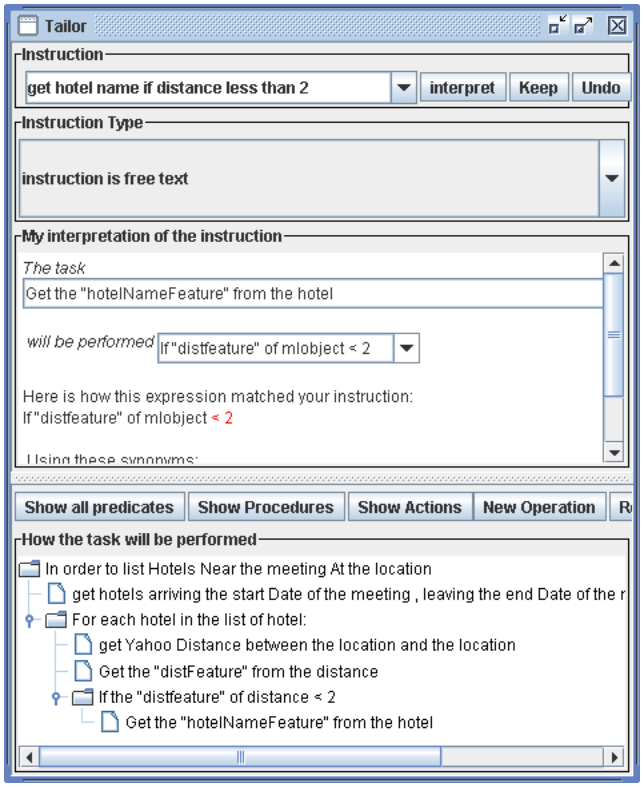

Figure 3: Tailor shows its interpretation of the user instruction and the resulting new procedure.

```
  [seq:
    [context: (startDate?meeting?Date4)]
    [context: (componentLocationCity?location?City5)]
    [context: (endDate?meeting?Date6)]
    [do:(getHotels?Date4?Date6?City5?hotelData6)]
    [do:(convertXMLToList?hotelData6?Listhotel7)]
    [forin:?hotel8?Listhotel7
     [seq:
      [do:(fetchAttribute?hotel8"hotelLocation"?Location8)]
      [do:(getYahooDistance?location?Location8?distanceData4)]
       [do:(extractObject?distanceData4?distance5)]
       [do:(fetchAttribute?distance5"dist"?DistanceMiles6)]
       [do:(stringToLeadingNumber?DistanceMiles6?Number7)]
       [select:
        (<?Number72)
        [do:(fetchAttribute?hotel8"hotelName"?Hotelname1)]
        (True)
        [succeed:]]]]]
)
```

Results and discussion

We used the combined system for 14 test problems relating to travel and organizing meetings that together required 11 information sources. Typical problems in these domains are: "List hotels within ?N miles of the ?Meeting that have the features (?F1, . . . ?Fn)", "Notify the attendees of ?Meeting of a room change" and "Who is attending ?Conference that

could present a poster at ?Time?". In these examples, we use "?" to denote a variable.

Agents were built for the information sources with EzBuilder, then PrimTL was used to predict the type of the inputs and outputs based on the content. The correct semantic type was often the top prediction, and for more than half of the sources, it was among the top four predictions. F-measures for the predicted type are at 0.8 or above for 7 of the 11 agents when we require the correct type to be PrimTL's first choice, and for 10 of the agents when the correct type must be one of the top four choices.

Working with correctly-labelled information agents, full procedures were successfully built using Tailor in each of the 14 test problems. The tool was also used successfully by another group to build these procedures and to define a new procedure in the same domain.

Explaining user instructions

The interpretation of user instructions in Tailor is similar to explanation-based learning. Given the user instruction, Tailor searches for a matching code fragment that plays a similar role to a proof. This fragment exploits the structure of the domain to provide a strong bias on the set of possible interpretations, exploiting the typing of the constituent procedures and of queries in the domain (marked with "context" in the SPARK fragment shown above).

Instead of a target concept, however, the context of the instruction provides a target location in the existing procedure that is to be changed. Rather than a proof, Tailor constructs a procedure fragment by composing procedures, queries and language constructs that provides an output value that can be used in the desired location. As an additional constraint, the procedure fragment should have a description that loosely matches the user instruction. Our current definition of a match requires that the words in the user instruction that specify the procedure fragment should appear in the language rendering of the fragment. Completing the analogy, the instruction is best viewed as part of the learning example, since it biases the search to produce results that it may not otherwise produce, just as the example in explanation-based learning leads to one of potentially many explanations of the target concept.

For example, in Figure 3, a condition is to be added to the step to retrieve each hotel name. The words are "distance less than 2". A natural candidate is to compare some number with 2, and then the fragment that generates the number should include the word "distance". Tailor finds the EzBuilder agent "getYahooDistance" satisfying this condition. Its inputs are locations, which match the input of the procedure or the location of the hotel. Tailor searches for a procedure fragment in which the inputs and outputs are matched on types, and this leads it to insert several auxiliary tasks as needed into the fragment. For example, it processes the result of the distance agent, which is a string in XML format, to extract the distance value before making the comparison. Given the instruction "if hotel distance less than 2", it uses an auxiliary procedure to extract the location of the hotel and uses this in the distance function.

In general, many potential procedure fragments must be generated very quickly to provide a seamless user experience. Tailor uses a dynamic programming approach to achieve this (Blythe 2005a). Our combination of demonstration and instruction allows rich procedural definitions to be learned from a very small number of examples combined with user input.

Related work

Our system is similar to information mediators, which provide a uniform access to heterogeneous data sources. To integrate information from multiple sources with a mediator, the user has to define a domain model (schema) and relate it to the predicates used by the source. The queries are posed to the mediator using the domain model and are then reformulated into the source schemas. The mediator dynamically generates an execution plan and passes it to an execution engine that sends the appropriate sub-queries to the sources and evaluates the results (Thakkar, Ambite, & Knoblock 2005). Our system is different on a number of levels: (1) it attempts to automatically model the source by inferring the semantic types of its inputs and outputs; (2) instead of a query, the system assists the user in constructing executable plans, or procedures that (3) may contain world-changing steps or steps with no effects according to the system's model.

Much of the work in procedure learning relies on user demonstration (Lieberman 2001; Oblinger, Castelli, & Bergman 2006; Lau *et al.* 2004). The user steps through a task and the system captures the steps, optionally variablizing and generalizing them. This is intuitive for users, but in some cases several examples may be required to find the correct generalization, and some aspects of a scenario may be hard to duplicate for demonstration. Tailor requires instruction, which forces the user to articulate the general terms of the procedure, but can be faster and require less infrastructure. One of the earliest such systems was Instructo-Soar (Huffman & Laird 1995) that learned rules for the Soar system.

Conclusions and Future Work

We have described an approach that combines learning by demonstration and instruction to allow end users to build procedures that compose information-gathering primitive procedures and customize their use. We combine the strengths of the approaches, incorporating demonstration for a task that the user would find difficult to describe verbally, and using instructions to add conditions or to fine-tune steps in ways that may be cumbersome to achieve through demonstration.

We are exploring several paths of future work. In type prediction we are exploring recognizing complex types that are composed of more primitive types, for example a date is composed of a day, month and year. This allows order-independent generalization as well as a reduction in the number of parameters for procedures and queries and a corresponding reduction in search complexity. In Tailor, we are exploring the use of a form of case-based reasoning to exploit information from previously-defined procedures. If

there are arbitrary choices to make in the parameters of a new step, for example, matching similar procedures that use such a step can provide strong hints for the best choices. We are also investigating how to incorporate stronger constraints into the search, for example that a hotel check-out date must come after the check-in date or that distances functions are symmetric.

Acknowledgments

We gratefully acknowledge valuable conversations with other members of the Calo task learning group at ISI, including Jose-Luis Ambite and Tom Russ. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

Blythe, J. 2005a. An analysis of task learning by instruction. *In Proceedings of AAAI-2005*.

Blythe, J. 2005b. Task learning by instruction in tailor. *Proceedings of the 10th international conference on Intelligent user interfaces*.

Huffman, S. B., and Laird, J. E. 1995. Flexibly instructable agents. *Journal of Artificial Intelligence Research* 3:271–324.

Lau, T.; Bergman, L.; Castelli, V.; and Oblinger, D. 2004. Sheepdog: Learning procedures for technical support. *Proceedings of the 9th international conference on Intelligent user interfaces*.

Lerman, K.; Minton, S.; and Knoblock, C. A. 2003. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research* 18:149–181.

Lerman, K.; Plangprasopchok, A.; and Knoblock, C. A. 2006. Automatically labeling the inputs and outputs of web services. *In Proceedings of AAAI-2006* 93–114.

Lieberman, H. 2001. *Your Wish is my Command*. Morgan Kaufmann Press.

Morley, D., and Myers, K. 2004. The spark agent framework. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems* 2:714–721.

Muslea, I.; Minton, S.; and Knoblock, C. A. 2001. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems* 4(1/2):93–114.

Oblinger, D.; Castelli, V.; and Bergman, L. 2006. Augmentation-based learning: combining observations and user edits for programming-by-demonstration. *Proceedings of the 11th international conference on Intelligent user interfaces* 202–209.

Thakkar, S.; Ambite, J. L.; and Knoblock, C. A. 2005. Composing, optimizing and executing plans for bioinformatics web services. *The VLDB Journal* 14(3):330–353.