Information Integration for the Masses

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

We have integrated three task learning technologies within a single desktop application to allow users to create intelligent software assistants to handle a range of office tasks. The learned task-related procedures are capable of gathering and integrating information from online sources, monitoring the performance of tasks over time, communicating with the user about that progress and taking world-altering steps, such as sending an email or booking a hotel. These technologies include a tool to create agents for programmatic access to online information sources, a tool that aligns the inputs and outputs parameters used by these sources to a predetermined ontology and a tool to create procedures that compose other procedures and queries, with iteration and branching, based on user instructions in text. We have integrated these tools within the CALO Desktop Assistant and used the integrated system to learn procedures to handle a variety of office and travel-related tasks.

Introduction

Intelligent information integration applications will soon assist users in planning and managing many aspects of everyday life, for example, planning travel or managing the purchasing of equipment. These tasks require user to combine information from a variety of heterogeneous sources and process the information as the user wishes. The information integration application may even be required to monitor information source and give user help where appropriate. For example, when planning a trip, the user may want initially to gather information about flights and hotels based on the her preferences and budget. After this initial stage, the application could monitor for changes in flight or hotel prices and changes in flight schedule.

In order to use such information integration systems effectively for a wide variety of tasks, the user must be able to easily integrate new information sources, and formulate and execute appropriate queries to these sources. Currently, accomplishing these steps requires sophisticated technical knowledge and familiarity with the workings of information integration systems, expertise that the average user cannot obtain without specialized training. Our goal is to automate the process of creating a new information integration application to a degree that it can be done by a technically unsophisticated user.

We have integrated three learning technologies within a single desktop application to allow users to create executable applications that integrate information from heterogeneous sources. The technologies are (1) EzBuilder - a tool to create agents for programmatic access to semi-structured data on the web, such as online vendors or weather sources, (2) PrimTL - a tool that assists the user in constructing a model of the source by mapping its input and output parameters to semantic types in a predetermined ontology and (3) Tailor - a tool that assists the user in creating executable queries, based on user instructions in text.

These learned queries are capable of gathering and integrating information, monitoring the performance of the application over time, communicating with the user about that progress and even taking world-changing steps such as booking a hotel. To the best of our knowledge, no previous tools have been developed that can allow users to create queries with such broad capabilities.

In this paper we describe our experiences integrating these tools within the desktop application developed by DARPA's CALO project[1]. We used our system to create queries for 14 sample problems chosen to test the system in realistic scenarios. The tool was used both by the developers and, independently, by a group at a different organization. The queries were subsequently executed within the Calo desktop assistant.

In the next section we describe the architecture for the learning system and introduce an example problem that we use throughout the paper. In the following sections we describe the component technologies - EzBuilder, PrimTL and Tailor - along with modifications made to allow for broad integration between components. Next we discuss the 14 test problems that were chosen and describe experiences in using the tool. We finish with a summary of future work.

Architecture and example problem

Figure 1 shows a simplified view of the overall architecture for the intelligent desktop assistant, including the task executor and the task learning components within the assis-

[1]http://www.ai.sri.com/project/CALO

tant. The Calo Desktop Assistant includes a variety of components aiding in understanding, indexing and retrieving information from meetings, presentations and online sources as well as calendar and email-based applications. Many of the actions taken by these components are coordinated by the task execution agent, SPARK (Morley & Myers 2004). SPARK interprets procedures defined in a language similar to PRS and manages concurrent execution of tasks, including cases where some may be active while others are waiting for some external response or information. Desktop components can cause complex procedures to be executed by posting goals with SPARK and users can execute the same procedures through an intuitive interface. The combination of EZBuilder, PrimTL and Tailor allows users to create complex queries, or procedures, that make use of on-line information that are executed by SPARK, allowing them to be invoked by the user or by other components within the intelligent agent.

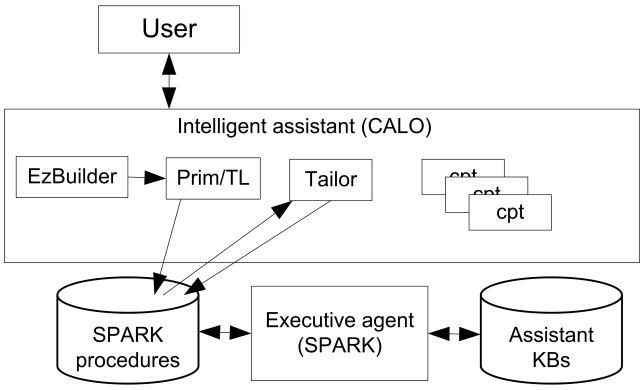


Figure 1: Learning agent architecture

We illustrate the problem with the following scenario. The user is planning travel and wishes to make use of a new online site for hotels and rates. First she builds a procedure to access the site, taking a city name as input and returning a list of hotels, with a set of features available for each hotel. Next, she creates a procedure that uses the first procedure to find a set of hotels within a fixed distance of a location available over a given time interval. This procedure can be called repeatedly with different locations, time intervals 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.

This example illustrates two motivations for our approach of providing access to online data sources within a general procedure execution framework. First, users put the data to use in several different ways. Second, the assistant's task involves more than simply data integration, in this case monitoring at fixed time intervals, sending email, and potentially re-booking the reservation.

Automating access to web-based information sources with EzBuilder

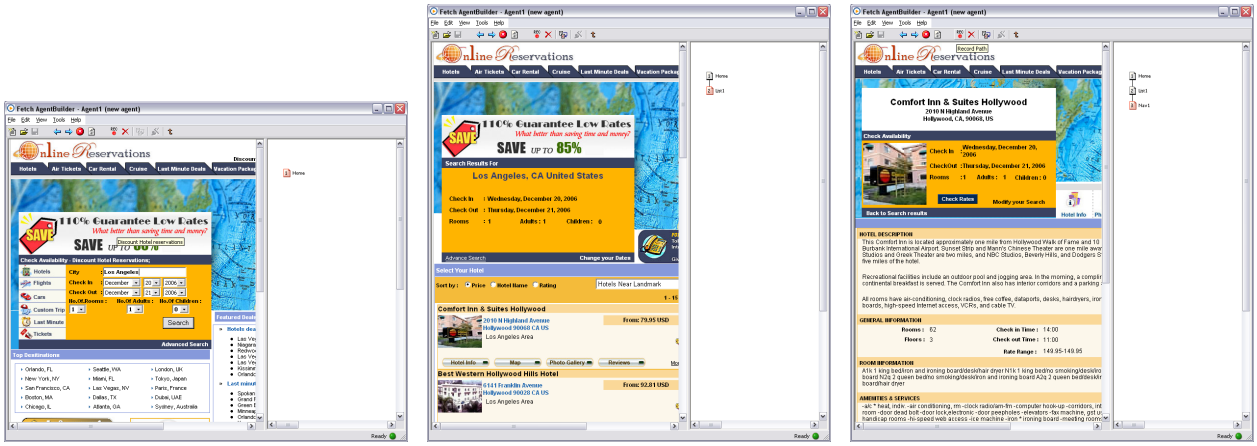
Most online information sources are designed to be used by humans, not computer programs. This design affects the way the site is organized and how information is laid out on its pages. The hotel reservations site shown in Figure 2 allows user to search for hotels available in a specified city on specified dates. After the user fills out the query form (Figure 2(a)) with her search parameters, the site returns a list of hotels (Figure 2(b)). The user then can then select a hotel from the list to obtain additional details about it (Figure 2(c)).

The Fetch Technologies' EzBuilder tool assists the user in creating agents that can automatically query and extract data from information sources. To build an agent for the online reservations site in Figure 2, 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.

Using this information, EzBuilder constructs a model of the site, shown in the right pane of Figure 2(c). Once the user has collected several detail pages by filling out the search form with different input parameters, EzBuilder generalizes the learned model to automatically download details pages from the site. During this step, EzBuilder also analyzes the query form and extracts the names the source has assigned to the various input parameters.

Next, the user specifies what data she wants from the site and trains EzBuilder to extract it, as shown in Figure 3. EzBuilder can extract a range of data structures, including embedded lists, strings, and URLs. For the reservations site, the user needs to extract for each hotel its name, address, city, state, price and a list of amenities offered by the hotel. After specifying the schema, the user trains the agent to extract data by specifying where on the sample page data examples are found. This is done by simply dragging the relevant text on the page to its schema instance. Once the sample pages have been marked up in this fashion, EzBuilder analyzes the HTML and learns the extraction rules that will quickly and accurately extract the required data from the pages (Muslea, Minton, & Knoblock 2001). If the pages are fairly regular, the user has to mark up one to three samples. 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. Generally, it is a good practice to leave some of the sample pages for testing extraction accuracy.

Once the user is satisfied that the correct extraction rules have been learned, she names the newly created agent - e.g., OnlineReservationZ - and deploys it to the server. This agent can now take user-supplied input parameters, query the source and return data extracted from the results pages. The extracted data is encoded within an XML string, with attributes defined according to the schema specified by the user. We wrote SPARK functions to extract specified data from the XML. These functions wrap Java methods implementing Xquery calls. If the agent returns a single tuple, requesting the attribute with a given name or label is suffi-



(a) (b) (c)

Figure 2: Examples of the query form and result pages from the hotel reservations web site

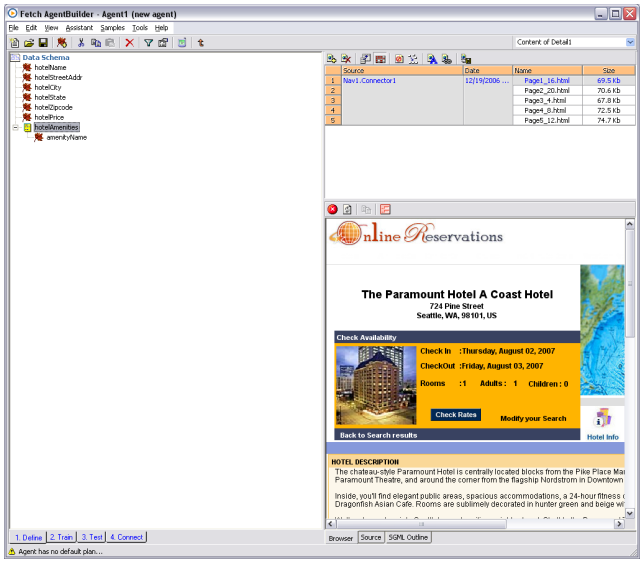


Figure 3: Schema definition for the hotel agent

cient. If it returns a list of tuples, the caller needs to iterate through the results, or access a given element, and return the named attribute. The names of attributes are contained within agent's registration string.

Modeling the agent with PrimTL

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. PrimTL also registers the agent as a fully typed primitive task, which can be automatically assembled into complex procedures by other CALO components.

In an earlier paper, we described a content-based classifier that 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, which simply identify the syntactic category to which the token's characters belong. The symbolic representation of data content by sequences of tokens and token types is concise and flexible. We can, for example, extend the representation by adding new semantic or syntactic types as needed.

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 later 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. We have developed a set of heuristics to evaluate the quality of the match, which includes how many of the learned token sequences match data, and how specific they are, how many tokens in the examples are explained by the model, and so on. We found that our system can accurately recognize new instances of known semantic types from a variety of domains, such as weather and flight information, yellow pages and personal directories, financial and consumer product sites, etc. (Lerman, Plangprasopchok, & Knoblock 2006).

PrimTL uses content-based classification methods to link

the agent's input and output parameters to semantic types in the CALO ontology. It reads data collected by EzBuilder, which includes the input data the user typed into the query form as well as data extracted from the result pages, and presents 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 4. The labels for each parameter are extracted from the form (for inputs) or schema names defined by the user (for outputs), although these metadata are not used for semantic labeling. The top scoring choice for each semantic type is displayed next to the parameter label. If the user does not agree with its choice, she can see the other choices, and select one of them if appropriate. Five of the seven input parameters corresponding to semantic types Day, Year and City are correctly assigned. However, the input parameter "ddinmonth" is incorrectly assigned to Speed. Similarly, no guess was made for the output parameter "amenityName", which has been assigned to the default base type clib:PseudoRange. In both cases, the user can correct the automatic guesses by manually specifying the correct semantic type. If the user is not familiar with the CALO ontology, she can launch it (through the "Show Ontology" button) and browse it to find the correct type.

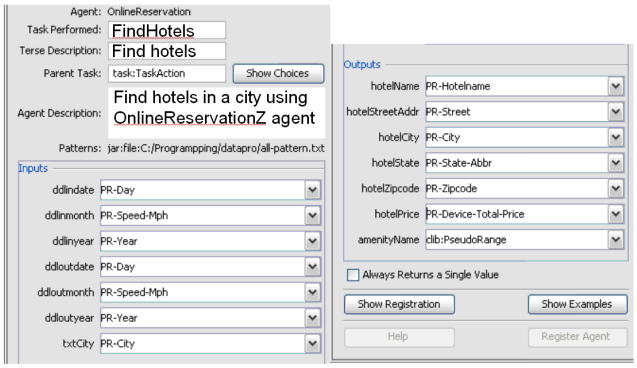


Figure 4: PrimTL's semantic mapping editor, which has been split in two to improve readability

In addition to specifying the semantic types of the input and output parameters, PrimTL requires the user to provide a short description of the agent's functionality and task performed. After this has been completed, the agent can be registered as a new primitive task with the CALO Task Interface Registry (TIR). The registration defines, within a single SPARK procedure, how the agent can be invoked, what its functionality is, what inputs it expects and what data it returns as output. TIR contains a library of these procedures which can be read in by CALO components.

Learning procedures with Tailor

Once the agent is registered as a primitive task, with input and outputs aligned to 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 three steps: *modification identification*, *modification detail extraction and modification 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.

A template for each modification type includes the fields that need to be provided and which words in the instruction may provide them. *Modification detail extraction* uses this information as input to search for modifications to procedure code. 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. Once mapped into a procedure modification, the condition may require several database queries and several auxiliary procedure steps. For example, a separate procedure that finds distances based on two addresses from an online source may be called before the numeric comparison that forms the condition. Tailor finds potential multi-step and multi-query matches through a dynamic programming search (Blythe 2005a).

Tailor's ability to insert multiple actions and queries is essential to bridge the gap between the user's instruction and a working procedure, and is used in most instructions. As another example, an EzBuilder agent is used to find the distance between the meeting and the hotel, requiring a street address and zipcode for each location. The user types "find hotel to meeting distance". Tailor matches the EzBuilder agent, and knows that meeting addresses can be found by querying the desktop database, while the hotel address can be found through a procedure that accesses the XML data returned from the hotel agent. Tailor inserts the correct code, and the user does not need to know these details. This capability depends on the correct alignment of the EzBuilder agents into the ontology, found by PrimTL.

In the final step, *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. The user may choose one of the remedies or they can ignore the problem: in some cases Tailor's warning may be incorrect because it does not have enough information for a complete analysis of the procedure, and in some cases the user may remedy the problem through subsequent instructions.

Prior to this implementation, Tailor had been used to mod-

ify existing procedures but not to create entirely new procedures. In order to provide this capability it was necessary to improve the efficiency of Tailor's search for matching code fragments. Tailor's initial search used dynamic programming on a graph of data types, with queries, procedures and iteration represented as links in the graph. We improved performance by dynamically aggregating groups of partially matched code fragments based on the user instruction. We also improved navigation of the results through grouping, highlighting and hierarchical menus, and gave the user more feedback and control over synonyms that are used in matching.

With these modifications, the example procedure (to alert the user if the price drops) can be built with four instructions. One iterates over the hotels and emails the user. One adds the condition that the hotel name matches the previously booked hotel and one checks that the price is lower than the previous booking. The last instruction sets the procedure to be called daily.

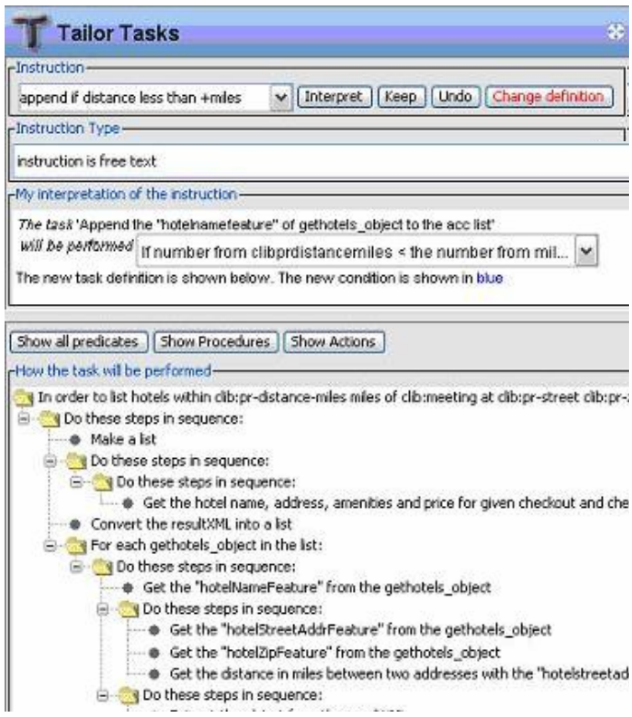


Figure 5: Tailor shows its interpretation of the user instruction and the resulting new procedure

Results

We used our learning system to build procedures for a variety of test problems in the office and travel domains. Typical problems in these domains are: "List hotels within ?Number miles of the ?Meeting that have the features (?Feature, , ?Feature)"; "Build a contact sheet of names, email addresses and phone numbers for the attendees of ?Meeting"; "Notify the attendees of ?Meeting of a room change"; "What purchases are due to be completed in the next month for ?Person?" and "Who is attending ?Conference that could present a poster at ?Time?" In these examples, "?" indicates presence of a variable.

We used EzBuilder to create agents for 11 information sources that provided the data necessary to solve the problems. These sources are listed in Figure 6, which also shows results of applying content-based classification algorithm to semantically label the input and output parameters used by these agents. The F-measure is a popular evaluation metric that combines recall and precision. Precision measures the fraction of the labeled parameters that were correctly labeled, while recall measures the fraction of all parameters that were correctly labeled. F1 refers to the case where the top-scored prediction of the classifier was correct, and in results marked F4, the correct semantic type was among the four top-scoring predictions. The correct semantic type was often the top prediction, and for more than half of the sources, it was among the top four predictions.

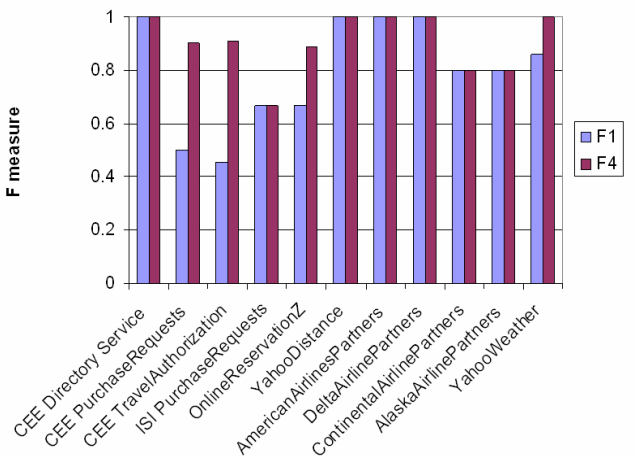


Figure 6: Semantic labeling results

Procedures were successfully built for each of the 14 procedures with Tailor using the information agents aligned with the ontology. However, the training requirements for users are currently higher than we would like: users need on the average ten hours of practice in order to build procedures of this complexity. The number of steps in the procedures ranges from 8 to 35, with an average of 18. The range reflects the range of complexities of the tasks. The number of instructions required to create the procedures ranged from 3 to 13, with an average of 7. Tailor is handling instructions that refer to several steps and conditions at once, with an average of 2.5 steps added per instruction. The number of alternative interpretations for instructions ranged from 1 to around a hundred, with 3 or 4 in most cases. In around 40% of cases, Tailor's first choice was the correct one. We are working on more support for harder cases, including a checklist of relations and procedures used to more easily remove unwanted matches.

Related Work

Our system is similar to an information mediator. Mediators provide a uniform access to heterogeneous data sources. To integrate information from multiple sources, the user has to define a domain model (schema) and relate it to the predicates used by the source. The user's queries, posed to the mediator using the domain model, are reformulated into the source schemas. The mediator then 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 known to the system.

Most work in procedure learning relies on demonstration from the user (Lieberman 2001; Oblinger, Castelli, & Bergman 2006; Lau *et al.* 2004). The user steps through solving a task and the system captures the steps of the procedure and generalizes 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 requires less set-up. One of the earliest such systems was Instructo-Soar (Huffman & Laird 1995) that learned rules for the Soar system.

Conclusions

We have described a system that uses learning technologies to assist a user in creating information integration applications. We evaluated the system on an interesting range of problems and found robust performance. With the aid of this system, even non-expert users were able to model new sources and construct queries to solve a wide range of problems.

Future work includes a more top-down integration, where the user starts by considering the overall capability to achieve, and creates information-gathering procedures as part of that, rather than our current bottom-up approach.

An interesting issue concerns the dynamic extension of the ontology based on the behavior of the information-gathering procedures. Suppose, for example, that the main ontology does not support 'zipcode', which is one of the outputs of the hotel agent and an input to the distance agent. Initially, the output can be mapped to a 'number', but more information is required for systems that compose procedures to know that this is a number that can be used with the distance agent. Such distinctions may not be needed in the rest of the system, however. In our current implementation we build a small auxiliary ontology of these terms that was shared between PrimTL and Tailor. In future work we will investigate how to support this dynamically for users.

In Tailor, we are working on the use of analogy to find similar procedures and offer ways to incorporate parts of those procedures into the one currently being built. This includes a 'smart drag-and-drop' that will recheck parameter bindings as a substep is copied from another procedure to the current target. This is useful since users often create procedures by copying from existing ones.

We are improving the performance of the semantic labeling technology in PrimTL by taking into account semantic type predictions for other data from the same information source. Although currently the user has to manually specify the task the agent is performing, we will integrate the technology to automate this step (Carman & Knoblock 2007). Improvements in EzBuilder will reduce the time it takes the train the agents by automatically extracting data from the information source.

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*.

Carman, M. J., and Knoblock, C. A. 2007. Learning semantic descriptions forweb information sources. *In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*.

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.