Journal of Universal Computer Science, vol. 14, no. 11 (2008), 1811-1837
submitted: 30/9/07, accepted: 25/1/08, appeared: 1/6/08 © J.UCS

Information Integration for the Masses

Jim Blythe, Dipsy Kapoor, Craig A. Knoblock, Kristina Lerman
(USC Information Sciences Institute, Marina del Rey, CA, USA
blythe@isi.edu, dkapoor@geosemble.com, {knoblock,lerman}@isi.edu)
Steven Minton
(Fetch Technologies, El Segundo, CA, USA
minton@fetch.com)

Abstract: Information integration applications combine data from heterogeneous sources to assist the user in solving repetitive data-intensive tasks. Currently, such applications require a high level of expertise in information integration since users need to know how to extract data from an on-line source, describe its semantics, and build integration plans to answer specific queries. We have integrated three task learning technologies within a single desktop application to assist users in creating information integration applications. It includes a tool for programmatic access to data in on-line information sources, a tool to semantically model them by aligning their input and output parameters with a common ontology, and a tool that enables the user to create complex integration plans using simple text instructions. Our system was integrated within the Calo Desktop Assistant and evaluated independently on a range of problems. It enabled non-expert users to construct integration plans for a variety of problems in the office and travel domains.

Key Words: Information extraction, web applications, assistants
Category: D.2.11, H.3.5, H.3.6, H.3.7

1 Introduction

To perform many everyday tasks, such as plan travel or manage equipment purchase, a user has to combine and process data from a variety of heterogeneous sources, which include web services and HTML-based on-line sources. For instance, when planning a trip, the user may wish to gather data about flights and hotels and filter them as follows: get the date and location of the meeting, find a convenient flight on those dates to that location, find the list of hotels in the city of the meeting with the required amenities, only consider hotels within three miles of the meeting site, only consider hotels with rates within government-allowed per diem, book a hotel with the best reviews, rent a car, and so on. Since these tasks are often repetitive, a user may wish to create an information integration application to perform them automatically. This application would even monitor information sources for changes in flight or hotel prices or changes in flight schedules and notify the user accordingly.

On-line information integration applications, or mash-ups that run inside a browser, have recently been popularised by companies such as Yahoo!, Intel, IBM or Microsoft. Users have created a variety of mash-ups, e.g., to display apartments available for rent on a map or show gas prices in a neighbourhood. Tools like Yahoo! Pipes and Microsoft Popfly enable non-programmers to create new mash-ups by reusing existing ones and modifying them to fit their needs. To create more advanced information integration applications, a user must be able to extract data from a source, specify its semantics [Knoblock et al., 1998] [Arjona et al., 2007], and formulate and execute the appropriate integration plans. Currently, these steps require sophisticated technical knowledge and familiarity with information integration systems [Thakkar et al., 2005], expertise that the average user cannot get without specialised training. It is an even more challenging task to create components and plans that can be reused by other users, because the users now need to share an agreement about the details of the semantics of sources and integration plans. Various initiatives, most notably the Semantic Web [Berners-Lee et al., 2001], have been advanced as a way to enable programmatic access to new sources, but they are slow to be adopted, and offer only a partial solution because information providers will not always agree on a common scheme.

Our goal is to automate the process of creating a new information integration application to a degree that it can be done by non-expert users. We have integrated three learning technologies within a single desktop tool to help users create applications that integrate information from heterogeneous sources, namely: EzBuilder, a tool to create information gathering procedures for programmatic access to on-line data sources; Semantic Mapper, 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 Tailor, a tool that assists the user in creating procedures, which are executable integration plans, based on short English instructions. The learned procedures are capable of gathering and integrating information, monitoring the performance of the application, communicating with the user about that progress and even taking world-changing steps such as booking a hotel. We believe that our tool is more flexible and general than recent task-learning systems like Plow [Allen et al., 2007] or Yahoo! Pipes.

In this article we describe our experiences integrating these tools within the desktop application developed by DARPA's Calo project, cf. http://www.ai.sri.com/project/CALO. The rest of the article is organised as follows: in Section 2, we describe the architecture of the learning system and introduce an example problem that we use throughout the article; in Sections 3, 4, and 5, we describe EzBuilder, Semantic Mapper and Tailor, respectively; in Section 6, we discuss the test problems that were chosen and describe several experiences in using the tool; in Section 7, we report on related proposals; our conclusions and future research directions are presented in Section 8.

2 Architecture and Example Problem

The Calo Desktop Assistant includes a variety of components that help understand, index and retrieve information from meetings, presentations and on-line sources, as well as calendar and e-mail-based applications. Many of the actions taken by these components are coordinated by the Spark agent framework [Morley and Myers, 2004], which is designed to scale well to large domains with a large number of executable

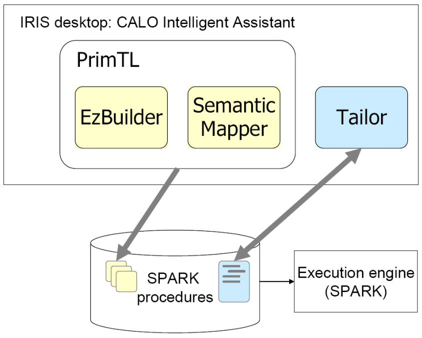


Figure 1: Diagram of the Calo architecture.

procedures while maintaining a simple semantics. Spark procedures describe how a task may be performed or what to do when a relation with a given type signature is added to its database. The body of the procedure may combine sequenced or unordered tasks with conditions, simple iteration and constructs for waiting until a condition holds. Steps in the body may also post goals to execute other tasks. One of Spark's strengths is that it can handle multiple threads, some of which may be waiting for information that comes from external sources such as e-mail. 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, Semantic Mapper and Tailor allows users to create complex integration plans that use on-line information, allowing them to be invoked by the user or by other components within the Calo Desktop Assistant. Figure 1 shows a simplified view of the overall architecture, including the task executor and the task learning components within the assistant. We illustrate the kind of problems in which we are interested with the following scenario. The user is planning a trip and wishes to use a new on-line site for hotel reservations. First he or she builds a procedure that takes a city name as input and returns a list of hotels, each with a list of amenities; next, he or she creates a procedure that uses the first one to find a set of hotels within a fixed distance of a location available over a given time interval; the user would like to be informed if the rate subsequently drops, so he or she creates another procedure that the assistant will automatically invoke daily to check the rate against the booked rate, and e-mail him or her if it has dropped. This example illustrates two motivations for our proposal: 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 e-mail, and potentially re-booking the reservation.

Figure 2: A query form and result pages.

3 Automating Access to Data Sources with EzBuilder

Many on-line information sources are designed to be used by humans, not computers. This design affects the way the site is organised and how information is rendered. For instance, the hotel reservations site in Figure 2 allows users to search for hotels available in a specified city on specified dates: first, the user fills in the query form and presses the search button; next, the site returns a list of hotels; and the user can then select a hotel from the list to get additional details.

EzBuilder assists the user in creating information gathering procedures that can

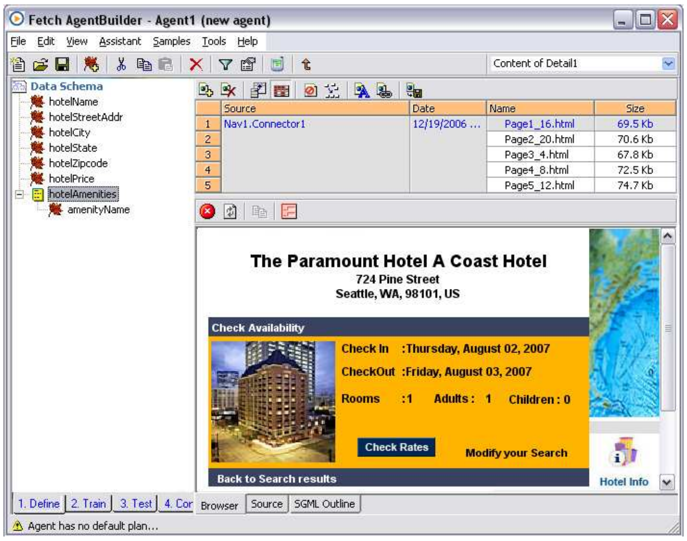


Figure 3: Scheme definition and data extracted from a page.

query and extract data from on-line information sources. To build an information gathering procedure for the previous on-line reservations site, the user demonstrates how to obtain the required information by filling in forms and navigating the site to the detail pages. Using this information, EzBuilder constructs a model of the site, cf. the right pane. Once the user has collected several detail pages, EzBuilder generalises the learned model to automatically download detail pages from the site. During this step, EzBuilder also analyses the query form and extracts the names the source has assigned to the various input parameters. Furthermore, EzBuilder can extract a range of data structures, including embedded lists, strings, and URLs. The user needs to specify the scheme and name the attributes to be extracted; next, he or she trains EzBuilder to extract data by showing it where data examples are located, which is done by simply dragging the relevant text on the page to its scheme instance, cf. Figure 3. Once the sample pages have been marked up, EzBuilder analyses the HTML and learns the extraction rules [Muslea et al., 2001].

Once the correct extraction rules have been learned, the user must name the newly created information-gathering procedure, e.g., OnlineReservationZ, and deploy it to a server that runs Fetch's AgentRunner platform. The procedure can take user-supplied input parameters, query the source and return data extracted from the web pages. To expose the results to the caller, all of the extracted data is encoded within a single XML string, with attributes defined according to the scheme specified by the user. We wrote generic Spark functions to extract this data from the XML. These functions wrap Java methods that implement XQuery calls. If the procedure returns a single tuple, requesting the attribute with a given name or label is sufficient; if it returns a list of tuples, the caller needs to iterate through the results, or fetch a given element and return the corresponding named attribute. The names of attributes are registered for the procedure by the Semantic Mapper, cf. Section 4.2.

EzBuilder is very flexible, and allows users to create procedures to extract data from a wide variety of web sources without requiring them to be proficient with HTML, scripting languages or even know how the data is rendered. However, it requires the user to mark-up pages. 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 information gathering procedure can learn general enough extraction rules. We are working on several tools that do not require the users to label data [Gazen and Minton, 2005] [Tuchinda et al., 2008]. Although such tools will not be able to extract data from as wide a range of web sources as EzBuilder, they will reduce the user effort in some cases.

4 Modelling Sources with Semantic Mapper

Although the newly created information-gathering procedure can now query information sources, it cannot be used programmatically by other Calo components because its data is not yet aligned with a common ontology. This is done by Semantic Mapper, which assists the user in modelling the source by linking its input and output parameters to semantic types in a common Calo ontology. Semantic Mapper also registers the procedure as a new primitive information-gathering task. This task can be automatically composed in complex procedures by other Calo components. This also enables information-gathering procedures created by one user to be reused by other users.

4.1 Semantic Labelling

We have developed a domain-independent data representation language that allows us to model the structure of the data returned by an information source as sequences of tokens or token types [Lerman et al., 2003]. Tokens are strings generated from an alphabet with character types such as alphabetic, numeric, or punctuation. We use the token's character types to assign it to one or more syntactic categories, e.g., alphanumeric (Alphanum), alphabetic (Alpha), capitalised (Caps), one-character capitalised (1Upper), two-character capitalised (2Upper), all caps (AllCaps), numeric (Number), one-digit number (1Digit), and so on. These categories form a hierarchy, which allows for multi-level generalisation, i.e., token 90210 can be represented by a specific token 90210, as well as general types 5Digit, Number, and Alphanum. These general types have regular expression-like recognisers, which simply identify the syntactic category to which the characters of the token belong. Our symbolic representation is concise and flexible since it can be extended with new semantic or syntactic types.

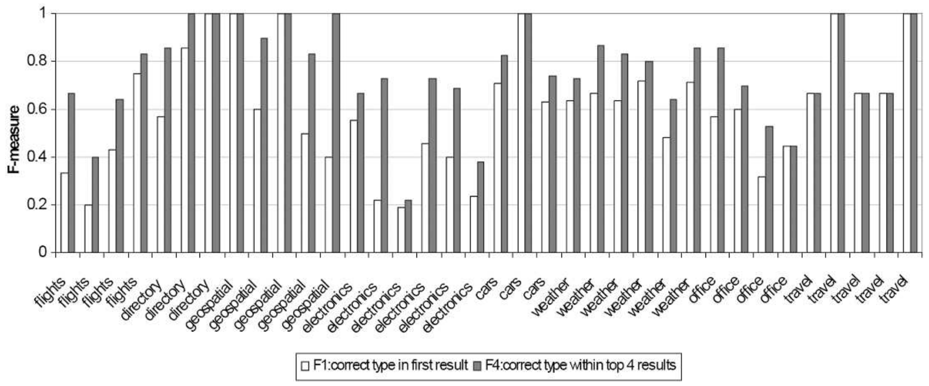


Figure 4: Results of semantic labelling.

We have accumulated a collection of models for about 80 semantic data types using examples extracted from a large number of online data sources by our group over the years. We can later use these models to recognise 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 a match, which includes how many of the learned token sequences match data, 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 recognise new instances of known semantic types from a variety of domains, such as weather and flight information, yellow pages, people directories, financial or consumer product sites, and so on [Lerman et al., 2004] [Lerman et al., 2006] [Lerman et al., 2007].

4.1.1 Simple Types

In [Lerman et al., 2003], we described an algorithm that learns the structure of semantic types from positive examples. It finds statistically significant sequences of tokens, i.e., those that occur more frequently than would be expected if the tokens were generated randomly and independently of one another. It estimates the baseline probability of a type's occurrence from the proportion of all types in the examples. For instance, if we are learning a description of street addresses, and have already found a significant sequence, e.g., the pattern consisting of the single token Number, and wish to determine whether the more specific pattern Number Caps is also significant. Knowing the probability of occurrence of Caps, we can compute how many times Caps can be expected to follow Number completely by chance. If we observe a considerably greater number of such sequences, we conclude that the longer pattern is also significant. An unintended consequence is that even a small number of occurrences of a rare token

Temperature	Sky	Windspeed	Visibility
32 F	Scattered Clouds	12 mph	7.00 mi
35.1Digit° F	Sunny and Windy	12 mph/19 kph	Unlimited
35 F	Cloudy	12 MPH	10.0 miles
36° F	Clear	9 mph	10.00 mi
36 F	Light Snow	9 mph/14 kph	10.00 Miles
39° F	Partly Cloudy	9 MPH	
39 F	A Few Clouds	22 MPH	
41 F	Overcast	2Digit mph	
70° F	Mostly Cloudy	2Digit mph/16 kph	
70 F	Fair	2Digit MPH	
2Digit° F		1Digit mph/0 kph	
2Digit F			

Latitude	Longitude	Latitude2	Longitude2
34.3Digit	-2Digit.0833	40 2Digit 00 N	40 2Digit 00 N
34.Number	-2Digit.9167	42 2Digit 00 N	42 2Digit 00 N
2Digit.617	-2Digit.0333	38 2Digit 00 N	38 2Digit 00 N
2Digit.067	-2Digit.2667	45 2Digit 00 N	45 2Digit 00 N
2Digit.6	-2Digit.4Digit	30 2Digit 00 N	30 2Digit 00 N
2Digit.6333	-2Digit.1Digit	37 2Digit 00 N	37 2Digit 00 N
2Digit.75	2Digit.183	2Digit 30 00 1Upper	2Digit 30 00 1Upper
2Digit.95	2Digit.324	2Digit 45 00 N	2Digit 45 00 N
2Digit.4Digit	2Digit.883	2Digit 24 00 N	2Digit 24 00 N
2Digit.3Digit	2Digit.983	2Digit 38 00 1Upper	2Digit 38 00 1Upper
2Digit.1Digit	2Digit.3Digit	2Digit 2Digit 00 S	2Digit 2Digit 00 S
2Digit.Number	2Digit.1Digit	2Digit 2Digit 00 N	2Digit 2Digit 00 N

Table 1: Sample patterns in the weather and geospatial domains. (Token types are underlined to distinguish them from regular text.)

will be judged significant. Furthermore, it is biased towards learning more specific sequences. For instance, when learning a model for Address from a set of addresses in which many are located at Main St and Elm St, the algorithm constructs (i) Number Alphanum St, (ii) Number Alpha St, (iii) Number Caps St, (iv) Number Main St and (v) Number Elm St. It eliminates sequence (i) because all the examples that match it also match the more specific sequence (ii); it eliminates sequence (ii) for the same reason; if sequence (iii) explains significantly more examples than the specific sequences (iv) and (v), it is kept and (iv) and (v) are deleted; otherwise, (iv) and (v) kept and (iii) is deleted. This tends to produce data models consisting of many token sequences, but the algorithm does not learn a complete model of the type, only of the common token prefixes. Table 1 shows a subset of the data model learned for different semantic types in the weather and geospatial domains. The learning algorithm learns, on average, about

30 patterns for each of the 80 semantic types in the domain model. Here only a few of the specific patterns are shown for each semantic type.

We can use the learned data models to recognise semantic types in new information sources. The basic premise is to check how well each semantic type describes the data based on the content of examples. We characterise how well a type matches data by computing a match score. In a nutshell, our ad-hoc scoring algorithm gives more weight to patterns that are longer or consist of more specific tokens. First, we assign a weight to the tokens, which increases with token's specificity. Thus, tokens such as AlphaNum have weight one, while specific strings have the highest weight (12 in this work). We calculate the score as follows: (i) calculate the pattern weight w_p as the sum of the token weights of a pattern's constituent tokens; (ii) calculate the weighted number of patterns matched as the sum of w_p for all patterns that match a prefix of an example; (iii) calculate the matching score, which assigns a higher score to more specific patterns and includes a penalty, e.g., tokens not matched by the pattern. The final score for a semantic type is the product of the weighted number of patterns matched and the mean matching score. The scoring algorithm returns the highest scoring F types (4 in this work), sorted by score, with the highest scoring match first.

Figure 4 shows the results of labelling several sources from a variety of information domains, namely: flights sources returned flight status information, weather sources returned current weather conditions at a particular location, directory sources returned contact information for specified people; other sites provided information about electronics equipment and used cars for sale; we also used several travel sites that provide hotels reservations and airlines partners information, and several sources from the office domain. The results are presented in terms of F-measure, a popular evaluation metric that combines recall and precision. $F1$ refers to cases in which the top-scored prediction of the semantic type was correct, whereas the correct type was amongst the top-four predictions in results marked $F4$. In most cases, the semantic labelling algorithm scores above 60% (for $F4$) and often above 80%. Some sources were difficult to label. For instance, most of the fields in the electronics domain contained only numeric tokens, e.g., 1024 for horizontal display size. The learned data model did not contain units information; therefore, it could not discriminate well between fields. In the flights domain, precision suffered because of non-standard capitalisation of text fields, e.g., FltStatus, and incompatible date formats in training and test data.

4.1.2 Complex Types

As the number of known semantic types grows, we are faced with a performance problem in both time and accuracy. For instance, Date is usually expressed as a combination of three simple types, i.e., Day, Month, and Year, that can be represented as single or double digits or even abbreviated names in the case of Month; furthermore, there are several ways to combine them, e.g., using "/" or ","; in addition, Year may be omitted, when the current year is implied; other complications include reversal of Month and

Flight	Date	Time	FltStatus
Alaska 562	06/12/2005	9:20 pm	on-time
United 1237	07/18/2005	8:30 am	delayed
Northwest 1123	06/13/2005	3:35 pm	on-time

Table 2: Examples of flight status data returned by a source.

Day, i.e., European versus American standards, and optional parts, such as the day of the week. The system may not be able to recognise a new example of a Date in a format it had not seen before. However, we can easily recognise the individual components of a date as Month, Day and Year, and we may be able to deduce that the unknown data type is Date, just because we have seen this combination used together in other sources.

We succinctly represent composite (or complex) data types like Date as a collection of simple types that can occur in any order. Thus, we keep track only of the observed probabilities of occurrence of simple types within the complex type. During the training step, we split the complex types into their constituent simple types, and label each fragment accordingly. The system then learns the associated occurrence probabilities of the simple types within the complex type. To keep the model simple, we do not allow for recursion, i.e., a complex type cannot be a component of another complex type. For instance, our system has learned the following occurrence probabilities from the known sources of dates (for clarity, complex types are prefixed by "CT-"):

```
CT-Date {
    Year Concrete 0.78 ()
    Month Abstract 1.0 (MonthName MonthNumber)
    Day Concrete 1.0 ()
    DayOfWeek Concrete 0.1 ()
}
```

According to this model, complex type Date contains Year with probability 0.78, which means that Date contained the type Year 78% of the time in the known sources. The reason that year is not always included in the date is that some sources assume the current year in the data they return. Month occurred in Date with a probability 1.0, Day with probability 1.0, and DayOfWeek with probability 0.1. The model further encodes the type Month as an abstract type whose instances can either be a MonthName or MonthNumber. Each of the simple concrete types is modelled by a set of patterns.

4.1.2.1 Training phase

Consider the data returned by a source about the arrival dates and times for different flights, cf. Table 2. To label the data, we split the columns with the complex data types into their constituent simple types based on the following delimiters: " ", ",", "/", ":",

CT-Flt:Airline	CT-Flt:FltNumber	CT-Date:MonthNum	CT-Date:Day	CT-Date:Year
Alaska	562	06	12	2005
United	1237	07	18	2005
Northwest	1123	06	13	2005

CT-Time:Hours	CT-Time:Minutes	CT-Time:Meridian	null:FltStatus
9	20	pm	on-time
8	30	am	delayed
3	25	pm	on-time

Table 3: Complex types broken into their constituent simple types. (CT-Flt is an abbreviation for CT-Flight.)

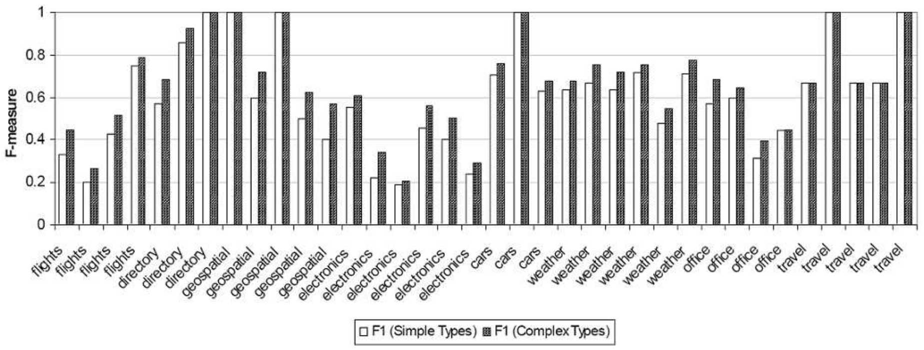


Figure 5: Comparison of semantic labelling performance.

"-", "(", and ")". We then label the columns as Complex-type:Simple-type. The resulting decomposed fields for the flight status source are shown in Table 3. The type null:FltStatus simply means that FltStatus is a simple type that does not belong to any complex types. We label multiple data sources in the above manner and count the number of occurrences of complex types and the number of occurrences of each simple type within the complex type. We compute the probability of occurrence of a simple type within a complex type from these figures. We then used the labelled examples to learn the patterns associated with simple types.

4.1.2.2 Testing phase

To assign semantic types to data using a model, we parse data examples into sequences of tokens using the same delimiters listed above. We attempt to match each example to a different combination of simple types, scoring each combination on the fly. We start by matching a sequence of tokens in the example to the learned model for simple types.

Once we find a match t_i, we compute its score r_i using the heuristics in Section 4.1.1 and continue looking for other matches starting at the next non-punctuation token.

This procedure generates a sequence M_j of matched simple types t_i and their scores r_i. We repeat this procedure iteratively, and generate all possible combinations of matched simple types. Next, we score how well each combination of matched simple types, M_j, fits a complex type of the domain model. A complex type C is represented as $\{s_1|p_1|c_1, s_2|p_2|c_2, \ldots, s_n|p_n|c_n\}$, where s_i is a component simple type, p_i is the probability of occurrence of the simple type s_i within the complex type C in the known sources, and c_i is a boolean indicator variable, with value 1 if $s_i \in M_j$, i.e., s_i was one of the matched simple types in M, and 0 otherwise. The match M is itself represented as a list $\{t_1|r_1|f_1, t_2|r_2|f_2, \ldots, t_m|r_m|f_m\}$, where t_i is the matched simple type with score r_i and f_i is a boolean indicator variable with value 1 if $t_i \in C_k$, and 0 otherwise. The score for $C = \{s_1|p_1|c_1, s_2|p_2|c_2, \ldots, s_n|p_n|c_n\}$ given $M = \{t_1|r_1|f_1, t_2|r_2|f_2, \ldots, t_m|r_m|f_m\}$ is calculated as follows:

$$S = \frac{\sum_{i=1}^{m} (f_i \times r_i)}{\left(\sum_{i=1}^{m} !f_i\right) + \left(\sum_{i=1}^{n} (!c_i \times p_i)\right)}$$

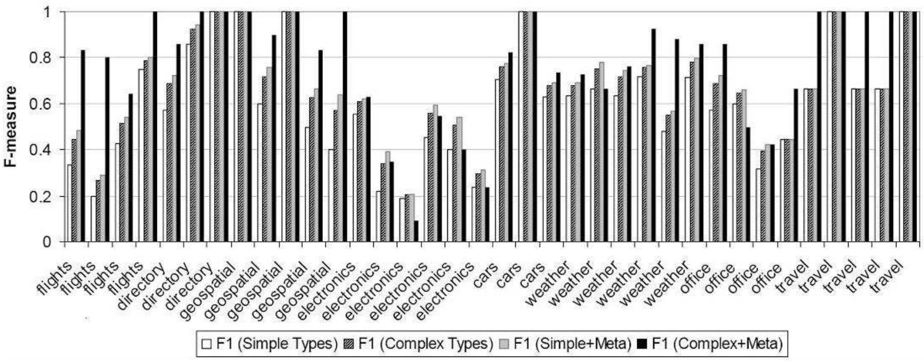
(The notation $!f_i$ means that the boolean variable is negated.) The scoring equation rewards the terms that match the complex type's constituents and penalises the terms from the complex type's constituents that were not found, as well as extra terms that were found but are not a part of the complex type. Each example generates a score for the complex type; they are then summed up over the examples and normalised over the number of examples. The type with the highest score most likely describes the data.

Figure 5 compares top-scoring semantic type prediction produced by the domain model with simple types only and the model with both simple and complex types. In all cases, the complex types model achieves at least as good a performance as the simple types model, and it leads to some improvement in most cases. it was not as great as we expected because introducing a large number of new simple types into the model led to poorer matches in the components of a complex field.

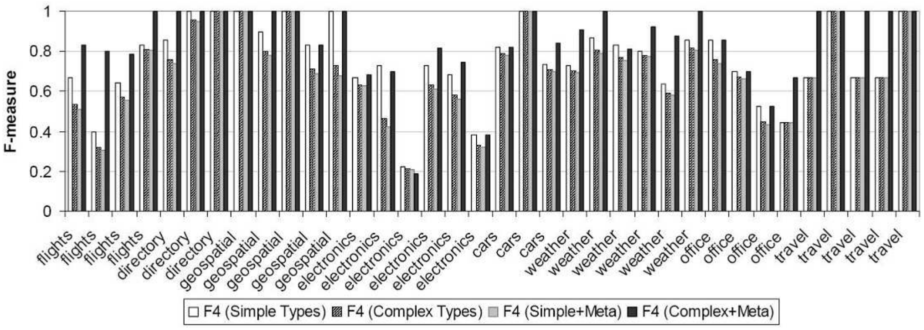
4.1.3 Meta-Labelling

Many simple types in the domain model have the same learned representation, e.g., StreetNumber and FltNumber are both represented as 1- to 5-digit numbers. Given a 4-digit number, it is difficult for the labelling algorithm to correctly identify its type. However, if the semantic types of other fields are known, then a better match can be found. For instance, if other fields being labelled are guessed to be Airline or Airport, then the 4-digit number is likely to be a FltNumber. The meta-labeller uses the likelihood of co-occurrence of types to produce a better label. From a labelled corpus, it computes the co-occurrence probability of each type with the other data types.

For instance, consider the data in Figure 2; for each field, we count the number of occurrences of the field within the known sources and the number of co-occurrences



(a) Correct type in the top prediction.



(b) Correct type within the top 4 predictions.

Figure 6: Results of using the meta-labeller.

of this field with other fields. Thus, in the example above, Airline was observed three times. The field Airline was also observed three times each with fields FltNumber, Date and FltStatus. We compute the probability of co-occurrence of each field with other fields and store them in the domain model. Our algorithm computes the co-occurrence probability of a type with all the complex types and their components. Here for simplicity we do not break up Date and Time into their constituent simple types. For instance, the data in Figure 2 leads to the following model:

```
Airline {
    FltNumber 1.0
    CT-Date 1.0
    CT-Time 1.0
    FltStatus 1.0
}
```

The co-occurrence probabilities can be used to enhance the matches produced by the semantic labeller. For each field, we generate a list of eight type predictions following the methodology described above. A permutation across all fields within a source gives all possible ways in which the top predictions can be combined. The co-occurrence probabilities are then used to evaluate each permutation by computing its likelihood. The scores of the individual matches are then modified so that they are the product of the old scores and the permutation likelihoods, which eliminates unlikely combinations, e.g., Airline and StreetNumber. Figure 6 reports on the results of applying the meta-labeller. Note that it dramatically improved semantic labelling results as compared to using simple types or complex types alone. With the exception of the especially difficult electronics domain, using a combination of complex type labelling with meta-labeller produced the best results. This combination raised the F-measure of the cases where the correct type was the algorithm's top prediction to at least 80% for 22 of the 36 sources. The correct type was amongst the top four prediction for 27 of the 36 sources.

4.2 Semantic Mapping Editor

Semantic Mapper uses semantic labelling to link a wrapper's input and output parameters to semantic types in the Calo ontology. It reads data collected by EzBuilder, which includes the values the user typed into the HTML query forms as well as data extracted from the result pages, and presents the user with a ranked list of predictions of the semantic type for each parameter. The semantic labelling step for the OnlineReservationZ site is shown in Figure 7. The labels for the parameters are extracted from the form (for inputs) or scheme names defined by the user (for outputs), although these names are not used in the labelling step. The top scoring prediction for each semantic type is displayed next to the parameter label. If the user does not agree, he or she can select another choice or type in the correct type. Note that Semantic Mapper automatically groups related fields into a complex type, e.g., the hotel street address, city, state and zip code were combined into a single complex field. The user is given a choice to ignore the automatic grouping with the check box next to it.

In addition to specifying the semantic types of the input and output parameters, Semantic Mapper requires the user to provide a short description of the information gathering task performed. After this has been done, the wrapper can be registered as a new fully-typed primitive task with the Calo Task Interface Registry. The registration defines how the procedure can be invoked, its functionality, what inputs it expects and what data it returns as output. The Task Interface Registry has a library of not only these primitive tasks, but other complex procedures created by Calo components. Once a primitive task is registered, it can be invoked by any Calo component.

5 Learning Procedures with Tailor

Once the information-gathering procedures have been registered, they are available through Spark and can be composed with other procedures to provide the desired func-

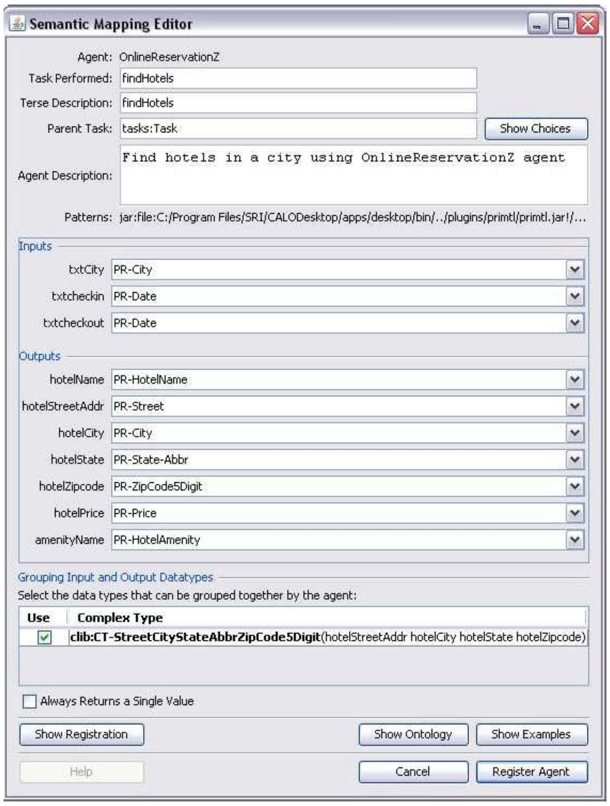


Figure 7: Registering a wrapper with Semantic Mapper.

tionality. For instance, the user may wish to create a procedure to periodically check hotels near a given location and send an e-mail message if the price drops by some threshold amount. Tailor allows the user to create procedures, which may be viewed as complex integration plans that use the newly registered tasks, by providing short instructions about the steps, their parameters and conditions under which they are performed. Tailor saves the user from needing detailed knowledge of the procedure syntax and ontology by mapping instructions into syntactically valid fragments of code that may compose several procedure calls, and describing the results using template-generated text [Blythe, 2005a] [Blythe, 2005b]. Tailor relies on the semantic labelling found by the Semantic Mapping Editor to find meaningful and useful combinations of procedures and queries in response to the user's instructions.Tailor uses two central modules, namely: instruction interpretation, which processes a user instruction and produces a set of potential modifications, and code fragment search, that takes parts of a user instruction and maps them to fragments of code that could match the text and that are syntactically correct given the current procedure being modified. Tailor's search also

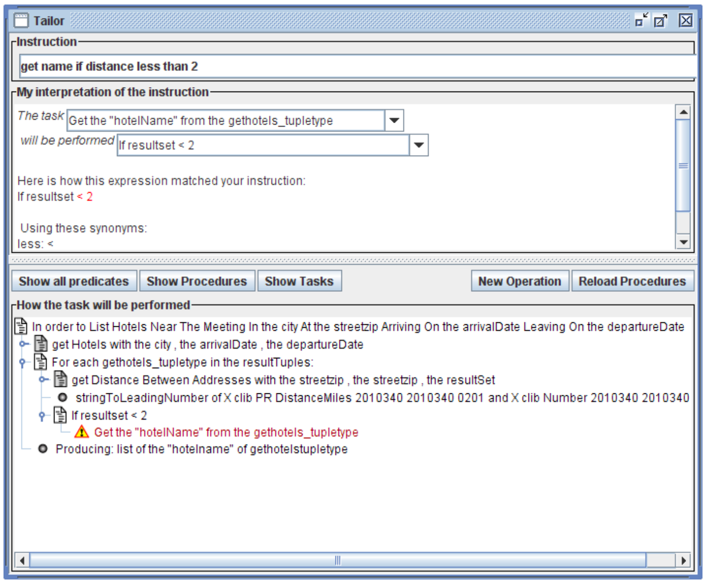


Figure 8: Tailor's user interface.

allows it to incorporate procedures using complex types without extra user effort.

Before describing Tailor in more detail, we present a brief example in which it is used to create a procedure to find available hotel rooms within a given distance of a location. The procedure combines two primitive tasks built using the tools described earlier: one that finds hotels given a city and dates for arrival and departure, and one that finds the distance between two addresses, each given by a street address and zip code. First, the user adds a step to find hotels into the new procedure; he or she just needs to give Tailor the instruction "find hotels", and it searches the Task Interface Registry for procedures that use these words. In this case, the hotel procedure shown in the previous section is found; it requires three inputs, i.e., the two dates and a city. Normally, Tailor would search for queries or procedures that produce variables of the required types using other input variables. However the procedure does not yet have any inputs, so Tailor suggests adding them as input variables. Next, the user chooses the hotelName field of the hotel record to be returned, from a menu. Since the procedure produces a list of hotels, Tailor automatically creates an iteration over the list in response to this choice. Finally, the user gives the instruction "get name if hotel distance less than 2 miles". Tailor recognises this as an instruction to make the step to return hotel names depend on some condition. It searches for hotel distance and finds the procedure to get the distance between two addresses. The word hotel is matched by using the street address and zip code from the hotel record returned by the procedure to find hotels. (Figure 8 shows the tool after matching this instruction. The window at the bottom

Instruction	Template
task before|after|while task	add-task-with-ordering
remove condition condition	delete-condition
remove task	delete-task
change argument arg in task to object	change-arg-in-task
task if|unless condition	add-condition-to-task
wait for condition then task	add-condition-to-task (rendered as a wait clause)

Figure 9: Tailor's set of modification templates and keyworded instruction templates.

shows a description of the steps using templates; some steps are omitted for the sake of clarity.) The full procedure contains steps to map data between different representations, e.g., from the string that represents a distance in a web page into a number that is used as a threshold, or from the complex address type into a simpler street–zip code form that is required by the distance procedure. In interpreting the second instruction, Tailor recognises that a test is being added, but also that several steps must be added to the procedure to enable it. While searching to match the phrase hotel distance, Tailor finds a sequence of steps to extract an address and uses it in the distance information gathering procedure; it also recognises that it can apply a projection between complex types to compose the steps. Tailor uses an efficient search method that allows it to compare many such compositions of code in real time.

5.1 Interpreting Instructions

Tailor helps a user create or modify a procedure 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 using key word analysis, e.g., 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 instance, would be classified as the latter. Figure 9 shows a representative list of the instruction key words used and the corresponding templates to make the modification; symbols task and condition in the text indicates that text found in this position in the instruction will be used by modification detail extraction to identify matching steps or conditions through search as described below. In [Blythe, 2005a], we show that a similar set of templates is complete in the sense that any procedure may be built using templates from that set.

A template for each modification type includes the fields that need to be provided and the words in the instruction that may provide them. Modification detail extraction uses this information as input to search for modifications to procedure code. For instance, 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

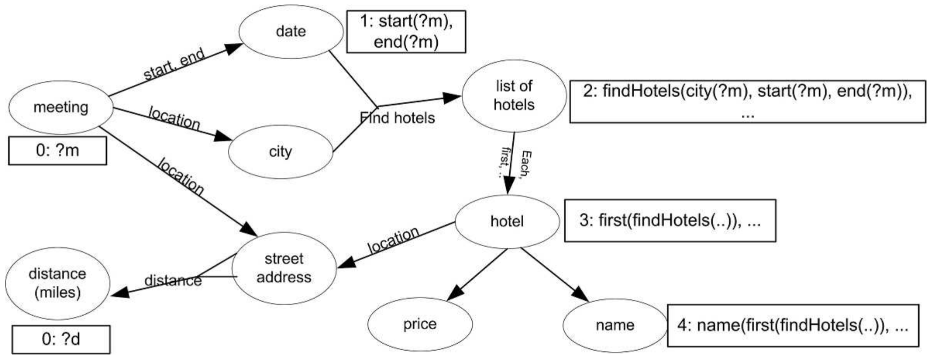


Figure 10: Part of the code search graph for expression "the name of the hotel".

condition may require several database queries and auxiliary procedure steps. For instance, a separate procedure that finds distances based on two addresses from an on-line 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]. This 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, suppose the user types "find hotel to meeting distance" before adding the hotels to the procedure. Tailor finds the distance procedure, 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 by the procedure that invokes the hotel wrapper. Tailor inserts the correct code, and the user does not need to know these details. This capability depends on the correct alignment of the wrappers into the ontology, found by the Semantic Mapping Editor.

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 instance, 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 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.

5.2 Code Fragment Search

Tailor uses a dynamic programming approach 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. Depending on the context of the request, the fragments may be constrained to be a single task or query, or a sequence of tasks and queries supporting a final condition. The type of the final value produced may optionally be constrained. For each of these variations, the basic search is performed in the same way.

Before learning begins, Tailor creates a graph whose nodes represent the data types present in the conceptual domain of the task, e.g. hotel and distance. A node exists for the input and output arguments of every procedure or relation in the domain, creating parametric nodes to represent lists of objects, e.g., list(hotel), and also for each complex type. The process is finite since it is driven by the finite set of procedures and predicates known to the primitive tasks. Each predicate or procedure is represented in the graph as a multi-link, linking the set of input types to each possible output type. Search is begun when the user indicates a new subtask to be added or condition to be applied at some point in the current task definition. Given an expression like "list each hotel name", Tailor searches for a step or condition that can be legally inserted at the point in the process and whose English description matches the words used, allowing for synonyms. During the search, Tailor generates candidate expressions by traversing the links in the graph of data types: each time a new set of objects becomes available at the set of input nodes for a link, we create a new object at each of the output nodes that represents the application of the related query or procedure to the input types. Whenever a new object is created, its description is checked against the search terms. Links are examined in sequence, so the first match found is the shortest in terms of links traversed. Relatively large code fragments may be built up in a few iterations; however, since intermediate results are cached at each node and re-combined, so the size of a fragment may grow exponentially in the number of iterations. The condition or step may refer to any variables that are bound at the point in the process where it is inserted, or to constants known to the system, i.e., the graph is initially seeded with objects that represent the programming context for the desired code fragment.

Figure 10 shows an example of searching to find matches for instruction "the hotel name"; only nodes that contribute to a match are shown. Ovals represent data types in the static graph, linked by queries or procedures that can create new terms; the boxes show terms that are created during search, along with the iteration at which terms are first created for the data type. Initially, two input variables are available, one for the distance in miles and one for the meeting, which is shown as the ?m term next to data type meeting. After the first iteration of search, terms are available at the date and city data type nodes, e.g., city(?m) represents the application of a query to find the city in which a meeting takes place. Two terms are available with data type date, and they correspond to the start and end dates of the meeting. On the next iteration, a list of hotel records is available, and it represents the application of a web procedure to find hotels based on a city, check-in, and check-out dates. Four such terms are available due to the possible combinations of dates, though only one leads to a reasonable procedure. This is the first term that matches any of the words in the user's instruction. Subsequent iterations convert the XML into a regular list of hotel records, iterate over them and retrieve their names. This term, which is a procedure fragment, matches all the words in the user's instruction and is shown as a possible match. Note that a new term should be propagated even if it does not match any part of the search string, because it may lie along a path to more complex terms that do match. This is the case for city(?m). However, once a term has been generated with a fragment that matches a subset of the search words, it is wasteful to propagate the same term and subset again, since all matches found using the fragment must have the same structure as those found with the original fragment. Instead, we add them to the set of fragments that have the same type and match set. If this set of fragments is later found to be part of a solution, Tailor generates the cartesian product of all such fragments as part of its solution set.

5.3 Complex Types in Tailor

Complex types are important both from the user's point of view and in Tailor's search algorithm. For the user, complex types can make a procedure more understandable by reducing the number of variables to interpret and putting them at a level that is closer to the user's viewpoint, e.g., the hotel's address rather than its street, city and zip code. For Tailor, complex types play an important role in reducing the number of parameters to consider during search; for instance, the procedure to find hotels available for a meeting requires a check-in date and check-out date, but the meeting object provides the meeting start and end dates. There may also be other procedures available to manipulate dates, e.g., to find the date some number of days before or after another date. Given the user instruction "get hotels for meeting" to add a step to a procedure that declares a meeting, Tailor searches for ways to fill the parameters of the primitive task for check-in and check-out date. Using primitive types, the hotel procedure has six parameters dealing with dates: a day, month and year for each of the check-in and check-out dates. Tailor has no information that these parameters are connected, and considers the two simple candidate options for each one, based on the start and end dates of the meeting. Thus it would find 2^6 potential solutions for the hotel procedure. When Tailor composes fragments of code, it bundles alternative paths that are equivalent in their input-output specification and in how they match the users instruction. Because of this, the 2^6 alternative steps using primitive types will not noticeably slow Tailor's search. However the user would have to choose between picking values for each of the six independent parameters, or look through all 2^6 alternative steps. Note that many of the alternative steps use irrelevant or even impossible dates, e.g., a combination of the day of a date

Question
List hotels within ?Number miles of a ?Location
List hotels within ?Number miles of a ?Location with ?FeatureSet
Rank hotels within ?Number miles of a ?Location based on ?FeatureSet
List hotels within ?Number miles of an ?Address
Rank ?CityList based on likelihood of weather delays on ?Date
List hotels within ?Number miles of an ?Address that give frequent flyer miles on ?Airline
List hotels in ?City with a swimming pool and free internet
Find the role ?Person plays in ?Institution
Build a contact sheet containing names, e-mail addresses and phone numbers for the attendees of ?Meeting
Notify the attendees of ?Meeting about a room change
List purchases due to be completed in the next month for ?Person
List people travelling to the same ?Location on the same day as ?Person
List people who will be at ?Conference on ?Date, based on travel plans
List grad students going to ?Conference

Table 4: The test questions used to validate the tool.

with the month of another. Using complex types, the two dates are represented as two parameters rather than six. There are two simple candidates for each parameter: the start and end times of the meeting; thus, Tailor suggests four possible steps to the user rather than 2^6, or the user can independently select values for two parameters rather than six.

Tailor is also able to offer higher-level help based on common operations with complex types. For instance, after a step that produces a complex type has been added to a procedure, Tailor provides menu options for returning or collecting individual components of the complex type, and assists the user with projections of complex types. This is done with the same general mechanism used for adding transformations and pre-processing steps needed for steps or conditions requested by the user. For instance, the hotel procedure returns an address for each hotel, as a complex type with five fields including the street, city, state and zip code. Another procedure that returns the distance between two addresses uses complex types that require only two fields for each address, the street and zip code. Tailor automatically adds code to create a new instance of the required input type from the hotel address when the user asks to compute the distance of the hotel. In some implementations of complex types in Spark, the larger address object can still be given to the distance-getting procedure as an input. When this is true, Tailor removes the extra steps from the procedure before sending it to Spark.

6 Validation

We used our system to build procedures for a variety of test problems in the office and travel domains. Typical problems in these domains include "List hotels within ?Number miles of the ?Meeting that have the features ?Feature, . . . , and ?Feature" or "Notify the attendees of ?Meeting of a room change". We used EzBuilder to wrap

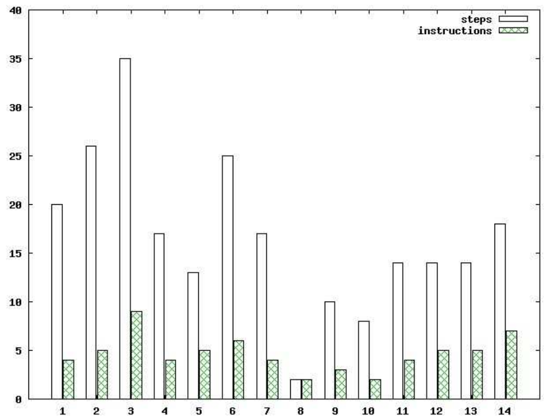


Figure 11: Number of steps and instructions required.

eleven information sources that provided the data necessary to solve these problems. The results of applying the semantic labelling algorithm to label the input and output parameters used by these sources are included in the results in Figures 5 and 6. For the sources used in the office and travel domains, the correct semantic type was often the top prediction, it was amongst the top four predictions for more than half of the sources.

The tool was used to successfully build procedures for the test questions in Table 4, with Tailor using the information gathering procedures 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 2 to 35, with an average of 16.6. The number of instructions required to create the procedures ranged from 2 to 9, with an average of 4.6. Tailor is handling instructions that refer to several steps and conditions at once, with an average of 3.6 steps added per instruction. This factor comes from a combination of Tailor's search, which may compose several steps in response to a query, and automated help for iteration, including gathering the results in a list. The number of alternative interpretations for instructions ranged from one to around 100, with three or four in most cases. In approximately 40% of cases, Tailor's first choice was correct. We are researching ways to provide more support for harder cases, including a checklist of relations and procedures used to more easily remove unwanted matches.

Furthermore, SRI International carried out an independent evaluation of our tool as part of its overall evaluation of the Calo system. Sixteen test subjects, software professionals from SRI with no specific training in information integration, participated in the study. Most of the subjects worked on the Calo project but did not work on EzBuilder, the Semantic Mapper or Tailor. In this study, the subjects used the combination of EzBuilder and Semantic Mapper to retrieve and model data relevant to answering the questions "find the role ?Person plays in ?Institution" and "list people travelling to ?Location on the same day as ?Employee". Each subject then used Tailor to answer three instantiations of each question. The answers were evaluated according to how closely they matched the known answers to these questions. For the first question, all subjects achieved perfect scores, exactly matching the expected answer. The subjects were able to generate correct answers for two of the three instantiations of the second question. The misses were explained by the fact that for one employee mentioned in the question instance, the travel location was entered as New York, NY and for another it was entered as New York City, NY. At this time, our system does not resolve references to handle such variations of target strings. Although this study is quite limited, we believe it offers support for the utility and ease-of-use for our tool.

7 Related Work

Our system is similar to information mediators. Such systems provide uniform access to heterogeneous data sources, and require the user to define a domain model (scheme) and to 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 schemes. The mediator then generates an execution plan and sends it to an execution engine, which sends the appropriate sub-queries to the sources and evaluates the results [Thakkar et al., 2005]. Our system is different since it attempts to automatically model the source by inferring the semantic types of its inputs and outputs; furthermore, instead of a query, the system assists the user in constructing procedures that may contain world-changing steps or steps with no effects known to the system.

Semantic modelling is similar to the schema matching problem that occurs when several databases must be integrated. Researchers have developed a number of methods to automate the schema matching process, including those that rely on the contents of data fields [Rahm and Bernstein, 2001]. In [Doan et al., 2001] and [Doan et al., 2003], the authors use machine learning to learn data source descriptions. In their system, a user labels data from a few sample sources, and the system then trains a suite of special-purpose classifiers to recognise some features of the data and the schema. Each of the classifiers have shortcomings, and require the use of a suite of learners to compensate. Although our approach is similar in spirit, we believe that it is more flexible since it relies on patterns that capture significant structure in data. This allows us to recognise fields regardless of their length, as well as frequently occurring specific data instances. We also have a unified probabilistic description of sources that allows us to handle both the hard and the soft constraints described in [Doan et al., 2001].

Most work in procedure learning relies on user demonstrations [Lieberman, 2001] [Oblinger et al., 2006] [Lau et al., 2004]. The user steps through solving a task and the system captures the steps of the procedure and generalises them. This is intuitive for users, but in some cases several examples may be required to find the correct generalisation, and some aspects of a scenario may be hard to duplicate for demonstration.

Tailor requires instructions, 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 and Laird, 1995], which learned rules for the Soar system.

A task learning system called Plow was recently shown to successfully learn executable information integration tasks by combining demonstration, natural language explanation and dialogue [Allen et al., 2007]. It relies on the DOM model to extract data from web pages. Our system can extract data from a wider variety of web sources because it tolerates noise and format inconsistencies by using machine learning tools. Furthermore, our system semantically models data sources and enables their reuse. Finally, our system offers the user the ability to learn and modify Spark procedures through short natural language instructions. These procedures may have been originally built by knowledge engineers or learned, by our own tool or by others. It is therefore able to modify a much wider range of existing procedures than Plow.

Recently, companies such as Yahoo! and Intel unveiled tools to help users create simple on-line information integration applications known as mash-ups, e.g., to combine apartments listings with maps. Yahoo! Pipes offers an intuitive visual interface to aggregate and manipulate web content. Users build a pipe (mash-up) by linking predefined widgets together, e.g., a widget may fetch an RSS feed and pass it on to another that filters it using key words. Although users can create new pipes by modifying the existing ones, they cannot reuse existing pipes within other pipes. The widgets that extract web content are similar to our wrappers, whereas widgets that manipulate content are similar to Tailor operations. Although there are a significant differences in how a pipe is constructed (visually) and how a procedure is created by Tailor (natural language), they share the same goal, i.e., composing and manipulating web information. Note, too, that any user can create a new information-gathering procedure with EzBuilder, but it is not clear how to create new widgets using Yahoo! Pipes. We also attempt to alleviate the need for the user to understand the semantics of data provided by web sites, while Yahoo! Pipes requires the user to model the on-line data source. Intel's MashMaker helps non-expert users create widgets that perform tasks such as extracting data from a web page or filtering. Although users need some degree of expertise to create widgets, non-expert users can then reuse them to create new integration applications. Unlike our system, in which the reuse of learned procedures is driven by their semantic definitions, in MashMaker, widget's reuse is driven by key words in its description and the context of the operation the user is performing. Users construct queries by example while browsing the data, rather than by explicit instructions as in Tailor. Since there is no empirical evaluation, it is difficult to say how well MashMaker works.

Karma is a recent system that simplifies mash-up creation by moving away from the widget paradigm [Tuchinda et al., 2008]. It seamlessly combines information retrieval, cleaning, modelling, and integration via a demonstration paradigm in which a user shows the system the data in which he or she is interested by copying it from a web page into a table. Karma relies on the DOM model to extract data from web pages, as well as to assign similar data to the same column of the table. Therefore, it cannot extract data from as wide a variety of web pages as EzBuilder. Unlike in our system, semantic modelling of data sources is done manually. Creation of data integration plans is driven by demonstration, as in MashMaker: the user assembles data from multiple pages into a single table, linking it through constraints imposed by the table. In contrast, our system creates integration plans by instruction.

8 Conclusions and Future Work

We have described a system that uses learning to assist a user in creating information integration applications. We evaluated the system on a range of real-world problems related by a common domain and found robust performance. With the aid of our system, users who had no previous experience with information integration tools were able model new sources, to extract data, and construct integration plans to solve a wide range of problems. Our system allows users to re-use learned procedures to build ever more complex and powerful information integration applications.

Future work includes a more top-down integration in which 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 behaviour of the information-gathering procedures. For instance, assume that the main ontology does not support zip codes, which is one of the outputs of the hotel procedure and an input to the distance procedure. 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 primitive task that extracts distance from a web source. 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 Semantic Mapper and Tailor. In future work, we will investigate how to support this dynamically. Regarding Tailor, we are working on the use of analogy to find similar procedures and offer ways to incorporate parts of them into the one currently being built. This includes a smart drag-and-drop feature that will recheck parameter bindings as a substep is copied from another procedure to the current target. Users often create procedures by copying from existing ones, and analogy will support this effort. It will also improve Tailor's rate of choosing the correct interpretation of an instruction by exploiting prior experience. We would like to improve semantic labelling performance on numeric types, which are currently treated as text strings. We also plan to automate primitive task recognition, based on the methods described in [Carman and Knoblock, 2007]. Concurrently, planned improvements in EzBuilder will reduce the time it takes the train wrappers.

Acknowledgements

We thank Tom Russ for implementing the Semantic Mapping Editor and parts of Tailor, and José Luis Ambite for providing useful feedback. This research is partially supported by the National Science Foundation (IIS-0324955), the Air Force Office of Scientific Research (FA9550-07-1-0416), and the Defense Advanced Research Projects Agency (FA8750-07-D-0185/0004). The U.S. Government is authorised to reproduce and distribute reports for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organisations or any person connected with them.

References

[Allen et al., 2007] Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., and Taysom, W. (2007). Plow: A collaborative task learning agent. In *Proceedings of the 22nd Conference on Artificial Intelligence*.

[Arjona et al., 2007] Arjona, J., Corchuelo, R., Ruiz, D., and Toro, M. (2007). From wrapping to knowledge. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):310–323.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, May:34–43.

[Blythe, 2005a] Blythe, J. (2005a). An analysis of task learning by instruction. In *Proceedings of 20th Conference on Artificial Intelligence*, pages 558–563.

[Blythe, 2005b] Blythe, J. (2005b). Task learning by instruction in Tailor. In *Proceedings of the 10th Conference on Intelligent User Interfaces*, pages 191–198.

[Carman and Knoblock, 2007] Carman, M. and Knoblock, C. (2007). Learning semantic descriptions for web information sources. *Journal of Artificial Intelligence Research*, 30:1–50.

[Doan et al., 2001] Doan, A., Domingos, P., and Halevy, A. (2001). Reconciling schemas of disparate data sources: a machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD Conference on Management of Data*, pages 509–520.

[Doan et al., 2003] Doan, A., Domingos, P., and Halevy, A. (2003). Learning to match the schemas of databases: A multistrategy approach. *Machine Learning Journal*, 50(3):279–301.

[Gazen and Minton, 2005] Gazen, B. and Minton, S. (2005). AutoFeed: An unsupervised learning system for generating webfeeds. In *Proceedings of the 3rd Conference on Knowledge Capture*, pages 3–10.

[Huffman and Laird, 1995] Huffman, S. and Laird, J. (1995). Flexibly instructable agents. *Journal of Artificial Intelligence Research*, 3:271–324.

[Knoblock et al., 1998] Knoblock, C., Minton, S., Ambite, J., Ashish, N., Modi, P., Muslea, I., Philpot, A., and Tejada, S. (1998). Modeling web sources for information integration. In *Proceedings of the 15th Conference on Artificial Intelligence*, pages 211–218.

[Lau et al., 2004] Lau, T., Bergman, L., Castelli, V., and Oblinger, D. (2004). Sheepdog: Learning procedures for technical support. In *Proceedings of the 9th Conference on Intelligent User Interfaces*, pages 109–116.

[Lerman et al., 2004] Lerman, K., Gazen, C., Minton, S., and Knoblock, C. (2004). Populating the semantic web. In *Proceedings of the Workshop on Advances in Text Extraction and Mining*, pages 33–38.

[Lerman et al., 2003] Lerman, K., Minton, S., and Knoblock, C. (2003). Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:149–181.

[Lerman et al., 2006] Lerman, K., Plangprasopchok, A., and Knoblock, C. (2006). Automatically labeling the inputs and outputs of web services. In *Proceedings of the 21st Conference on Artificial Intelligence*, pages 93–114.

[Lerman et al., 2007] Lerman, K., Plangprasopchok, A., and Knoblock, C. (2007). Semantic labeling of online information sources. *International Journal on Semantic Web and Information Systems*, 3(3):36–56.

[Lieberman, 2001] Lieberman, H. (2001). *Your Wish is my Command*. Morgan Kaufmann.

[Morley and Myers, 2004] Morley, D. and Myers, K. (2004). The Spark agent framework. In *Proceedings of the 3rd Conference on Autonomous Agents and Multiagent Systems*, pages 714–721.

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

[Oblinger et al., 2006] Oblinger, D., Castelli, V., and Bergman, L. (2006). Augmentation-based learning: Combining observations and user edits for programming-by-demonstration. In *Proceedings of the 11th Conference on Intelligent User Interfaces*, pages 202–209.

[Rahm and Bernstein, 2001] Rahm, E. and Bernstein, P. (2001). A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350.

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

[Tuchinda et al., 2008] Tuchinda, R., Szekely, P., and Knoblock, C. (2008). Building mashups by example. In *Proceedings of the 2008 International Conference on Intelligent User Interface*, pages 9–18.