TrIAs: Trainable Information Assistants for Cooperative Problem Solving

Mathias Bauer & Dietmar Dengler
DFKI
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
{bauer,dengler}@dfki.de

Abstract

Software agents are intended to perform certain tasks on behalf of their users. In many cases, however, the agent's competence is not sufficient to produce the desired outcome. This paper presents an approach to *cooperative* problem solving in which an information agent and its user try to support each other in the achievement of a particular goal. As a side effect the user can extend the agent's capabilities in a programming-by-demonstration dialog, thus enabling it to autonomously perform similar tasks in the future.

1 Introduction

Software agents are intended to autonomously perform certain tasks on behalf of their users. In many cases, however, the agent's competence might not be sufficient to produce the desired outcome. Instead of simply giving up and leaving the whole task to the user, a much better alternative would be to precisely identify what the cause of the current problem is, communicate it to another agent who can be expected to be able (and willing) to help, and use the results to carry on with achieving the original goal.

The *user* of a system can certainly be expected to have some interest in obtaining a useful response, even at the cost of having to intervene from time to time. As a consequence it seems rational to ask her for help whenever the system gets into trouble. By enhancing the communication between user and agent by a *training phase*, the agent's capabilities can be extended such that similar problems in the future can be prevented and the problem-solving process becomes more efficient.

This paper presents an approach to *cooperative* problem solving in which a software agent and its user try to support each other in the achievement of a particular goal. By demonstrating the solution of a subproblem, the user extends the agent's capabilities and thus enhances its usefulness for *all* potential users. As this *training mode* based on the paradigm of *programming by demonstration* supports various levels of expertise, even naive users can add to the overall performance of the system such that a flexible adaptation to a rapidly changing environment can be achieved.

In the case presented here, a trip planner tries to satisfy a user's requests using information from various Web sites. The central challenge of this system is the cooperation with the user whenever the system cannot satisfy an information request.

The rest of this paper is organized as follows. The next section introduces the overall system architecture and describes the trip planner application scenario. Section 3 describes the knowledge representation, query planning, and information integration aspects of TrIAs. Section 4 discusses the interaction between user and system in the attempt to oversome deficiencies in the system's model of an information source. Finally, Sections 5 and 6 discuss related work and summarize the results, respectively.

2 The TrIAs Framework

In the following, we will introduce the TrIAs architecture for *Trainable Information Assistants* which in this case has been instantiated by an application plug-in to an *Internet Travel Arrangement Assistant* (ITA) (see Figure 1). In this tour arrangement scenario the user and ITA cooperate in planning and assembling a trip.

The three core components of the architecture are the *application module* instantiated by the *trip planner* ITA and its knowledge sources, the *Information Broker* responsible for satisfying the requests for and controlling the access to information, and the *Information Extraction Trainer* (IET) helping the broker to bridge potential gaps in its knowledge about how to gather requested information. The planner and IET share the same user interface which is a standard *web browser*. The architecture supports interchanging the roles of the user and the Broker as an information *consumer* and an information *provider*. Regarding the consumer role the user specifies the desired trip using the interface provided by the trip planner. The trip planner is implemented as a constraint-based process able to handle different means of transportation, integrate appropriate accommodations, etc. Using all these data it incrementally refines the trip specification considering the user's preferences.

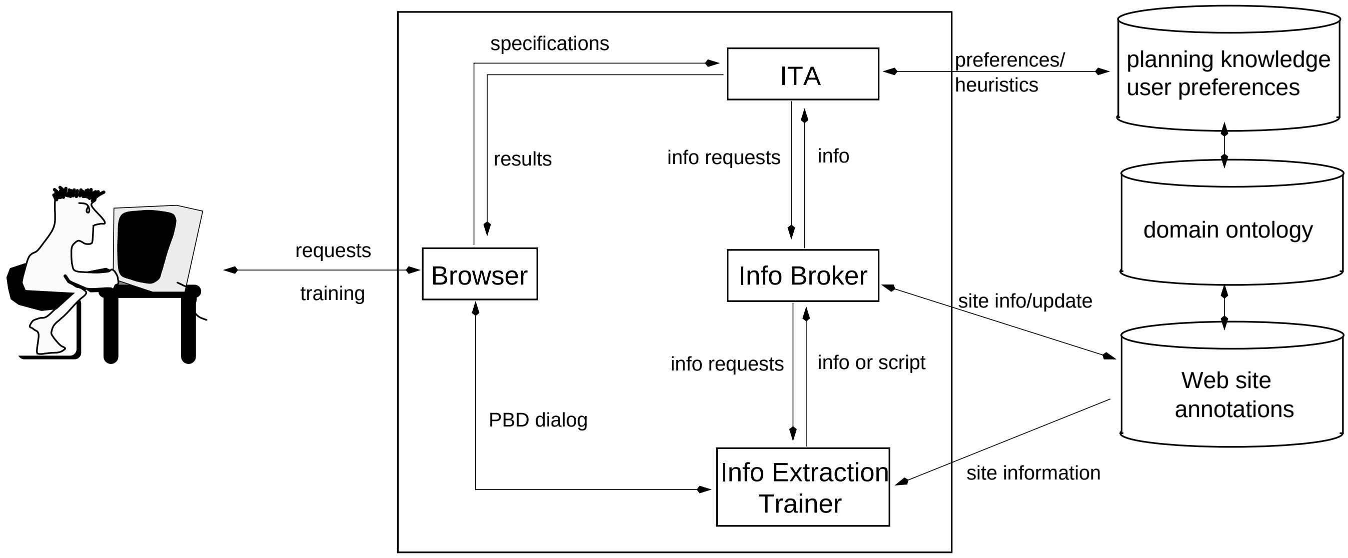

Figure 1: The TrIAs Architecture

Whenever the trip planner has an information gap that cannot be filled using its current domain knowledge—because the data are highly dynamic and currently out-of-date or are not locally available—it sends appropriate information requests to the Information Broker. This module maintains a database containing detailed specifications of the various information sources to be used, including schematic representations of the *types* of information contained (in terms of a shared, domain-dependent ontology as described in Section 3.1) and operational descriptions of how to actually extract them. These are scripts implemented in the Web query language *HyQL* (see Section 3.3). The Information Broker executes the *HyQL scripts* required and hands over the results to the application module, thus satisfying the original information request.

From time to time, however, the Broker's search for appropriate information is unsuccessful. Depending on the type of situation, the system reacts in one of several possible ways. If the Web site to be accessed cannot be reached, the broker simply tries to satisfy the application's information request using an alternative information source. The interesting case occurs when the intended web site could actually be accessed, but the associated HyQL scripts did not produce the desired outcome. A simple sort of syntactic *type checking* is used to discover that, e.g. a *date* was extracted when actually a *time* was expected (see Section 3.2).

The most frequent reason for this is a modification in the Web site layout. In this case the original information request, the HTML document under consideration, and the HyQL script responsible for the failure are handed over to the *Information Extraction Trainer* that starts a *programming by demonstration* (PBD) dialog with the user in which the roles of information provider and consumer are exchanged.

The objective of this dialog is the generation of a working script for the modified document—to be used for future information extraction—or at least the identification of the relevant information to be provided to the application. While the former enables the system to successfully deal with this Web site—at least until the next significant modification takes place—the latter only allows the *current* process to be carried on. Section 4 will give a detailed account on how user and system interact to deal with failed information requests and cooperatively try to achieve the overall domain goal.

3 Knowledge Representation and Query Planning

Domain-dependent *ontologies* play a key role in the knowledge representation of TrIAs. Defining a set of concepts and their interrelationships, they provide a common language to be used by all system components as well as a means for the integration of information from previously unrelated sources. As a consequence they have to be used in

- the formalization of the (domain) planning operators of the application as well as the domain-specific user preferences; in the above example, this refers to the preconditions and effects of the trip planning operators (e.g. constraining the set of admissible train rides) and users' preferences regarding airlines or transportation means.
- the description of Web sites; here concepts defined in an ontology are used to describe what kind of information can be found at a particular site (this and additional important aspects regarding information sources will be discussed more thoroughly in Section 3.2).

- the formulation of information requests (that is, application and Information Broker use the ontology as a common language for message exchange) and query planning. Often more than one information source has to be addressed in order to satisfy an information request. The query planning process sketched in Section 3.2 selects these sources and determines how and when they have to be queried.

3.1 The Use of Ontologies

Ontologies provide a modeling scheme for information sources that—when agreed upon by a number of agents—provides the basis for the exchange of messages and information regarding a certain domain. Following the Ontobroker approach [Decker *et al.*, 1999], a variant of frame logic is used to represent the various concepts and their interrelationships like subclassing, part-of relationships, and attribute declarations.
The expression

```
Flight[ carrier =>> Airline;
        departure =>> CityCode;
        destination =>> CityCode;
        travel_date =>> Date;
        departure_time =>> Time;
        arrival_time =>> Time;
        number =>> Flight_Number;
        ...].
```

for example defines the concept `Flight` to have attributes `carrier` of class `Airline` (to be defined elsewhere in the ontology), `departure` and `destination` of class `CityCode` and so on.

The formal language made up of the so defined concepts and relationships can then be used to describe, e.g. the contents of an information source in a way similar to a classical data base schema (see also Section 3.2).

When using these sources to answer information requests of the application system, a number of technical subtleties have to be dealt with. Assume, for example, that the arrival time of a flight could be extracted from an information source S_1 that now has to be used to search for a connecting train in a source S_2. While S_1 represents time in the American 12-hour format including an AM/PM indicator, S_2 makes use of the European 24-hour format.

To deal with this problem of *syntactic variants* of identical pieces of information, TrIAs extends the standard ontologies by a means to represent classes *intensionally.* Figure 2 depicts that part of the trip ontology encoding the definition of the `Time` concept with two attributes. `Time` has two subclasses `24h_Time` and `12h_Time`. The latter possesses an additional attribute `am_pm`. Each of these classes in turn subsumes a number of subclasses `rec_24_i` and `rec_12_i`, respectively.

The interesting point is the fact that the latter refer to "recognizers". A recognizer is a simple program that performs a kind of "type checking", i.e. it tests whether or not a given string (e.g. extracted from some online document) contains a piece of information in a particular format. As a side effect the corresponding attributes (`hr` and `min` in the `Time` example) are instantiated with their respective values in the *canonical representation* specified for each concept. Note that the way recognizers are implemented as simple DFAs accepting a limited formal language, they can just as well be applied to produce any syntactic variant from a given instantiation of the canonical representation.[1]

Examples of recognizers include those for time (in 12- or 24-hour format with varying separators, see Figure 2), date, phone numbers etc. The above problem of re-using the 12-hour time from source S_1 to query server S_2 expecting a 24-hour format is solved by applying the corresponding recognizer, say `time_12_1`, to the output of S_1, and the appropriate 24-hour recognizer (specified in the annotation of this Web site, see Section 3.2), say `time_24_3`, to the canonical representation of this time instance to produce the input as required byS_2.

3.2 Web Site Annotations

The most important knowledge source available to the Information Broker is a database containing annotations of Web sites to be used to answer information requests of the application module. For each Web site captured it contains

1. a *site contents schema* representing the types of information that can be found at a particular source in terms of the concepts defined in the current ontology,
2. a number of *querying schemes* reflecting the various ways an information source can be queried (these play an important role in the query planning process sketched below),
3. statistical data about typical response times to be used in query optimization, and
4. procedural descriptions of how to extract relevant information from an online document.

While components 1 and 2 initially have to be provided by a knowledge engineer (possibly the user herself), the statistics are automatically updated after each query. The final aspect of the Web site annotations, however, the so-called *wrappers* for information extraction encoded in the Web query language *HyQL* (see Section 3.3), are maintained and updated in a dialog between system and user whenever the need arises. Section 4 will explain in some detail when and how this happens.

Example: The contents of the ITN Web site[2], for example, is described by the schema

```
itnflight( Flight.carrier: Airline,
           Flight.departure: CityCode,
           Flight.destination: CityCode,
           Flight.travel_date: date_us_2,
```

[1] As recognizers can be used to "enumerate" all instances of the class they are associated with, they form their intensional representations.

[2] `www.itn.com`

```
    Time[ hr =>> Hour; min =>> Minute; ].

24h_Time :: Time.          12h_Time[ am_pm =>> AM_PM_Indicator; ] :: Time.
time_24_1 :: 24h_Time.     time_12_1 :: 12h_Time.
   ⋮                          ⋮
time_24_n :: 24h_Time.     time_12_m :: 12h_Time.
```

Figure 2: Part of an ontology including syntactic variants.

```
Flight.arrival_time: time_12_3,
Flight.departure_time: time_12_3 )
```

which indicates that the data to be found here are organized in tuples containing (part of) the information describing an instance of class `Flight` as defined above.[3] Note that for date and time entries the particular syntactic variant in terms of a corresponding recognizer is given (compare Section 3.1).

In order to use such a Web site as an information source, it is necessary to specify in which way(s) it can be queried to produce the intended outcome, i.e. to provide the information required. The *querying schemes* mentioned above describe which kinds of information have to be provided by the querying agent (in terms of attributes of the site contents schema as discussed above) and which attributes can be instantiated by actually performing this query and evaluating the resulting document.

A querying scheme < `+, +, +, u, -, -` > for the ITN Web site, for example, reflects the fact that querying this source by providing concrete values for

- `Flight.carrier`,
- `Flight.departure`, and
- `Flight.destination`

produces a document from which

- `Flight.arrival_time` and
- `Flight.departure_time`

can be extracted using the appropriate HyQL scripts specified. However, no information regarding `Flight.travel_date` can be found in this case (compare the representation of source "capabilities" in the Information Manifold [Levy *et al.*, 1996] for a similar mechanism).

To summarize, a '+' marks a position in the site contents schema for which the querying agent has to provide a concrete instantiation, while a '-' stands for an attribute that can be instantiated after the query. A 'u' denotes an attribute that is not involved in this particular kind of query.

Note that in general more than one querying scheme representing alternative ways to address an information source can be specified.

[3] An expression like `Flight.carrier: Airline` refers to the `carrier` attribute of an instance of class `Flight` being a member of class `Airline`.

The Information Broker treats these querying schemes as planning operators with preconditions (information marked with a '+' has to be available) and effects (information marked with a '-' becomes available). Given an information request and a representation of the information already available to the system (playing the roles of goal and initial state in planning terminology, respectively), a query plan is generated by combining the operators in a progressive way. This query planning process exploits statistical access information to identify promising paths and supports partial replanning in case of failed attempts to access an information source (similar to [Knoblock, 1995]). The details go beyond the scope of this paper.

3.3 The HyQL Query Language

Our approach of a web query language called HyQL combines but also extends features found in related language designs. HyQL's main purpose is to operationalize basic information gathering processes in the sense of document and information selection using guided search and filtering based on content and structure. Elaborating XML's [xml, 1997] flexible linking and referencing concept we integrated features in order to be able to specify robust queries.[4] Since we focus on applications in the context of trainable information assistants where the user defines information gathering actions in cooperative interaction with the system, user-defined abstractions can be integrated into queries in the form of macro operations. This is helpful because HyQL is the target language used in *programming by demonstration*-sessions to represent the intermediate and final hypotheses about the user's intended information selection concept (see Section 4).

HyQL considers the web as a computable dynamic graph structure where the nodes are static or dynamically generated documents and the edges are the links between them. Navigation and search in the graph structure is specified using path regular expressions based on flexible document quantification considering time-bounded forms of existential and universal quantification, respectively.

The documents themselves are represented as parse tree structures in a canonical form, which means that

[4] Robustness refers to the property of a script to be insensible to certain types of modifications of a document like adding commercial banners to an online document or changing its layout.

Figure 3: PBD-based wrapper specification.

some obvious faults in documents are repaired, optional start or end-tags are added, and additional annotations like "word" or "number" are integrated.

There are many ways to characterize the position of specific elements E in a document tree: by specifying constraints on the paths from the root to the relevant subtrees, constraints on properties of E, specifying a context or "*landmark*" of E and E's relative position to it, etc. *Collections*[5] play a key role in the specification of a position. There are collections of subtrees where the root is a specific tag, has specific attributes, the attributes are set to specific values, or the leaves match specific patterns. All collections can be accessed as a complete list, or indexed forwards or backwards. In addition to the single element and all-collections, selections of intervals are also possible: fetching a prefix or suffix of an all-collection, or specifying two single border elements and automatically all elements between them.[6]

HyQL allows the definition of *macros* which are parameterized abstractions of HyQL-expressions. They can e.g. be used to wrap user-defined annotations of document parts and consequently to specify new kinds of collections (see script `FLIGHT` in Figure 4).

Besides the integration of *recognizers* (cf. Section 3.1) HyQL also allows to specify restrictions that integrate structural properties of documents. The following example shows a selection script realizing the extraction of the attribute `arrival_time` applicable in the context of the site-specific relation `itnflight` (see also Section 3.2):

This script selects the 'arrival time' of a flight relative to the position of the 'destination' in a specific table in a document specified by its URL (see the sample document in Figure 3). The table is constrained to contain rows which must be of type `FLIGHT` which is a macro selecting a row of a table where the first data field must contain a boldface string which can be classified as the name of a carrier—`Airline` is a recognizer which is just able to do that. Using the context of a `FLIGHT`-conform row, we can select the text of the first element below a `BR`-tag in the third field of this row and check it with the recognizer `itncity` to validate it as a specific city format. Using this element as a landmark we can select 'arrival time' as the text of its second predecessor after validating it by the recognizer `time_12_3`.

Some further examples about the use of HyQL can be found in [Bauer and Dengler, 1998].

[5] Collections are sets of elements of a specific type, specific structure or content, or with specific attributes. All collections share the same access methods.

[6] We distinguish different modes of intervals whichever is the kind of subtrees spanning the collection, e.g. all lines of a table vs. a table as a whole.

```
select arrival_time =
       text(second predecessor of lm)
from lm =
  select first successor of first BR in third TD
         in first element FLIGHT in first TABLE t
         from document d in http://www.itn.com/...
         where destination = text(lm) and
               itncity(destination) and
               FLIGHT applicable to t
where time_12_3(arrival_time)

define macro FLIGHT
as TR constrained by t=text(first B in first TD)
                     and Airline(t)
```

Figure 4: HyQL script for the extraction of attribute `arrival_time`.

4 Interaction with the user

Assume that—as was mentioned in Section 2—the Information Broker failed to satisfy an information request from the current application system ITA. The system now initiates a dialog with the user during which she is first asked to identify those portions of the contents that are relevant to the query under consideration. The basic idea is to translate the user's selection into an executable access method in terms of HyQL and iteratively refine it by combining additional hints given by the user and system heuristics that serve to identify "landmarks" to be used for robust navigation. The objective is to synthesize a script that enables the system to autonomously deal with this Web site in the future.

Before the training process starts, the document currently under consideration is enriched by additional HTML features (tags and attributes) serving to mark single words, numbers, special characters and so on. These are used to better characterize the user's navigation through the document. Furthermore, JavaScript functions allowing text portions to be highlighted and uniquely identified are used to enhance the interactive functionality of the document. It is important to note that these changes remain invisible to the user, that is, the appearance of the document within the browser will be unaffected. Figure 3 depicts the situation where an access method for the arrival time of a flight for the ITN server is to be defined.

Parsing this modified document yields a tree that serves as the main data structure for the training process. The user's selection of text portions can be uniquely mapped onto this parse tree. Learning the concept hidden in the example given by the user then amounts to translating the corresponding subtrees into operational access procedures in terms of HyQL.

For the rest of this section we will use the example depicted in Figure 3 where the user is asked to identify the arrival time of a flight.

4.1 Wrappers in HyQL

In the TrIAs framework a wrapper, i.e. a HyQL script describing how to extract a certain type of information from a document, basically consists of two parts. The first one is a characterization of the document contents initially marked by the user—the data to be extracted—in terms of syntactic features like HTML tags surrounding it and the position of the corresponding subtree within the parse tree of the whole document. As we want to generate wrappers that are as robust as possible against certain types of modifications of the document layout—like adding commercial banners or reformatting part of the text—we have to add some more abstract elements to this description of the user's selection. Identifying "landmarks"—salient features of the document supporting the unique identification of some part of its contents—and adding them to the description constrains and focuses the navigation within the document at some level of abstraction by ignoring obviously irrelevant details.

In the example, each select button on the left can serve as a landmark to identify the table to its right containing the information of the various flights. Describing a flight this way—as a table following a select button—safely ignores anything between the top of the document and the first occurrence of a select button (this simulates the behavior of an HLRT wrapper as defined in [Kushmerick *et al.*, 1997]) and is thus insensitive to any modifications made there.

In principle any syntactic feature of a document can serve as a landmark: a string formatted in some particular style or containing some substring, a picture, or even a single HTML tag. Additionally, references to the context of the current information request can be used as a kind of "semantic" landmarks. In our example, it is possible to e.g. characterize the position of the arrival time relative to the string containing the destination of the current flight, in this case, Boston.

4.2 The Training Dialog

While the first step—the selection of the relevant document parts—is clearly the user's task, the identification of reasonable landmarks and stepwise improvement of the wrapper takes place in a cooperative dialog between user and system that will be sketched in the following.

After the user has marked the relevant document parts and the system has computed an initial characterization in terms of the associated subtrees within the document parse tree,[7] the user can

- provide further examples (and possibly counterexamples) of the concept to be learned (thus enabling the system to look for discriminating features),
- give additional hints facilitating the identification of landmarks,
- accept or reject landmarks suggested by the system,

[7]For details of how this translation takes place see [Bauer and Dengler, 1998].

- ask the system to present its current hypothesis and criticize its performance, and
- terminate the dialog at any time.

The system, on the other hand, can

- ask for additional examples,
- identify the next positive example within the current document by itself,
- suggest landmarks using a number of heuristics and process the user's feedback,
- classify the user's selection using *recognizers*, and
- present its current hypothesis to the user.

Note that this dialog does not follow a fixed protocol. Instead, each of the two partners can take turns suggesting additions to or modifications of the current wrapper.

In order to reduce the user's workload as much as possible the system has a number of heuristics at its disposal enabling it to search for landmarks by itself. Thus the user's task is often limited to accepting or rejecting the system's suggestions rather than having to look for landmarks by herself. These heuristics include searching for graphical objects or especially formatted strings in the immediate neighborhood of the selected document parts, counting the numbers of rows and columns in a table, testing the applicability of certain recognizers, or finding a reference to an already instantiated feature of the current information request context (for similar heuristics see [Ashish and Knoblock, 1997]).

At any time the system can present its current hypothesis to the user by executing the corresponding HyQL script step by step, each time highlighting the document portion being selected or displaying a simple natural language translation of its code.

In the example depicted in Figure 3 the system tries to find a landmark within the limited context of a `FLIGHT` (which corresponds to the highlighted table containing the description of flight Lufthansa 422). As the selected text is contained within the same table cell (the same element TD within the HTML encoding) as the string representing the already known destination of this flight, it is used to produce a HyQL script like `FLIGHT` depicted in Figure 4. Alternatives would have been to use the starting point (Frankfurt) or simply counting the row and column of the relevant cell within the table. The latter, however, is sensitive to a simple reordering of the table. As a consequence heuristics including numerical position descriptions are applied with low priority only.

4.3 Further Aspects of the Training Dialog

If a particular concept—e.g. the description of a flight at the ITN Web site—occurs more than once in a single document, the user can greatly facilitate the system's task by explicitly marking its first and/or last occurrence or restricting the search area by simply marking obviously irrelevant document parts like the header including a number of graphics. In this case a wrapper consists of a method to find the first occurrence of some interesting concept within a document and a description of how to find the next one.

Macros play an important role in the cooperative definition of wrappers. On the one hand they simplify the training process by restricting the relevant context and thus the search space for navigational aids like landmarks which is advantageous both for the system and its user. Note that the macro `FLIGHT` as defined in Section 3.3 was used just for this purpose in the training dialog sketched above (see also Figure 3). On the other hand they faciliate communication between system and user, as new wrappers can be explained in terms of already known concepts.

Finally it should be mentioned that the above training method can also be used to create wrappers for distributed documents interconnected by hyperlinks. Treating these links like ordinary edges in the document parse tree, wrappers can be synthesized that gather information by visiting more than one Web site.

5 Related Work

Instantiations of TrIAs are in the tradition of information integration systems like the Information Manifold (IM) and Ariadne (see [Levy *et al.*, 1996] and [Knoblock *et al.*, 1998], respectively). While these concentrate on efficient generation and execution of query plans and the integration of information from previously unrelated sources, TrIAs adds the aspect of cooperative problem solving by getting the human user involved in both training the system and gathering information.

Other differences can be found, e.g. in the basic assumptions regarding the information sources available. IM, for example, makes use of detailed source descriptions that explicitly represent constraints regarding the information contained at a particular site (e.g. information about cars *with price > 20000$*) in order to optimize the query process. TrIAs, on the other hand, relies on user interaction to correct obviously wrong source descriptions or add missing details whenever the need arises. This, in turn, is in contrast to *automatic* approaches to capture the contents of an information source in terms of users' categories like ILA [Perkowitz and Etzioni, 1995]

[VanLehn, 1987] uses the term "felicity conditions" to describe in which way the trainer of a learning system should structure the input in order to facilitate the learning task as much as possible. In the situation of information gathering as it typically occurs in the TrIAs framework, the roles are distributed in a slightly different way. As the system actually asks for help, the user's own "felicity" has to be taken into account. So the system has to find a balance between acquiring enough information to produce a robust wrapper and annoying the user by involving her in a lengthy training dialog.

Current wrapper generation approaches like [Hsu, 1998] and [Muslea *et al.*, 1998] use 'landmark grammars' working on the tokenized document string as a means to specify wrappers. Since we use a canonical parse tree structure extending the sequential token representation we are able to specify more robust wrappers integrating more abstract landmark patterns. The *wrapper induction* methods in [Kushmerick *et al.*, 1997], [Hsu, 1998], and [Muslea *et al.*, 1998] mainly aim at automatically constructing content extraction procedures for Web sources. They usually require more than a few examples to work and a user-interactive approach with online integration of new information sources is not adequately supported.

The HyQL language incorporates some ideas from other approaches: both WebSQL [Arocena *et al.*, 1997] and W3QL [Konopnicki and Shmueli, 1995] provide sophisticated constructs for navigation on the Web, but they lack an integrated model of parsing documents and selecting specific parts of them. SgmlQL [SgmlQL, 1997] is a nice SQL-style programming language for manipulating SGML (HTML) documents but lacks any navigation constructs. XML [xml, 1997] allows with its linking concept based on XML parse trees a precise specification of resource locations, especially within documents.

6 Conclusion

This paper presented the TrIAS architecture for trainable information assistants. It described how a problem solver—e.g. a trip planner—making use of information available on the WWW is embedded into an information integration framework that relies on user interaction in order to recover from failed information gathering actions.

A programming-by-demonstration dialog is used to correct the system's misconceptions regarding contents or layout of an information source, thus providing the basis for successful information gathering in the future. The expressiveness of the Web query language *HyQL* used to implement the required information extraction *wrappers* as well as the incorporation of structural and semantic constraints in the characterization of relevant information and the use of recognizers make the access procedures learned this way very robust against modifications of document layout and structure.

Future work will include the improvement of the PBD component with the aim to provide robust wrappers in a more intuitive way and the integration of an ontology management component.

References

[Arocena *et al.*, 1997] G. Arocena, A. Mendelzon, and G. Mihaila. Applications of a web query language. In *Proceedings of the 6th International WWW Conference*, Santa Clara, California, 1997. see also `www.cs.utoronto.ca/~websql/`.

[Ashish and Knoblock, 1997] N. Ashish and C.A. Knoblock. Semi-automatic Wrapper Generation for Internet Information Sources. In *Proceedings of the Second IFCIS Conference on Cooperative Information System (CoopIS)*, 1997.

[Bauer and Dengler, 1998] M. Bauer and D. Dengler. TrIAs – An Architecture for Trainable Information Assistants. In *Proceedings of the Agents98 Workshop on "Agents in Interaction"*, 1998. available at `www.dfki.de/~bauer/publications/agents98-ws.ps.gz`.

[Decker *et al.*, 1999] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: The Very High Idea. In R. Meersman et al., editor, *Semantic Issues in Multimedia Systems*. Kluwer, 1999.

[Hsu, 1998] C.-N. Hsu. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Workshop on AI and Information Integration*. AAAI, 1998. see also `http://www.isi.edu/ariadne/aiii8-`wkshp/proceedings.html9.

[IJCAI-95, 1995] *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[Knoblock *et al.*, 1998] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Muslea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 1998.

[Knoblock, 1995] C. Knoblock. Planning, Executing, Sensing, and Replanning for Information Gathering. In IJCAI-95 [1995], pages 1686–1693.

[Konopnicki and Shmueli, 1995] D. Konopnicki and O. Shmueli. W3qs: A query system for the world-wide web. In *Proceedings of VLDB Conference*, 1995. see also `www.cs.technion.ac.il/~konop/w3qs.html`.

[Kushmerick *et al.*, 1997] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 729–735, Nagoya, Japan, 1997.

[Levy *et al.*, 1996] A. Levy, A. Rajaraman, and K. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd VLDB Conference*, 1996.

[Muslea *et al.*, 1998] I. Muslea, S. Minton, and C. Knoblock. Learning wrappers for semistructured, web-based information sources. In *Workshop on AI and Information Integration*. AAAI, 1998. see also `http://www.isi.edu/ariadne/aiii8-`wkshp/proceedings.html9.

[Perkowitz and Etzioni, 1995] M. Perkowitz and O. Etzioni. Category translation: Learning to understand information on the internet. In IJCAI-95 [1995], pages 930–936.

[SgmlQL, 1997] Sgmlql: SGML Query Language. see `www.lpl.univ-aix.fr/projects/SgmlQL/`, 1997.

[VanLehn, 1987] K. VanLehn. Learning one Subprocedure per Lesson. *Artificial Intelligence*, 31:1–40, 1987.

[xml, 1997] Extensible markup language (XML): Part 2. linking. W3C Working Draft July-31-97, `http://www.w3.org/TR/WD-xml-link`, 1997.