TrIAs – An Architecture for Trainable Information Assistants

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 a software 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.

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 him 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.

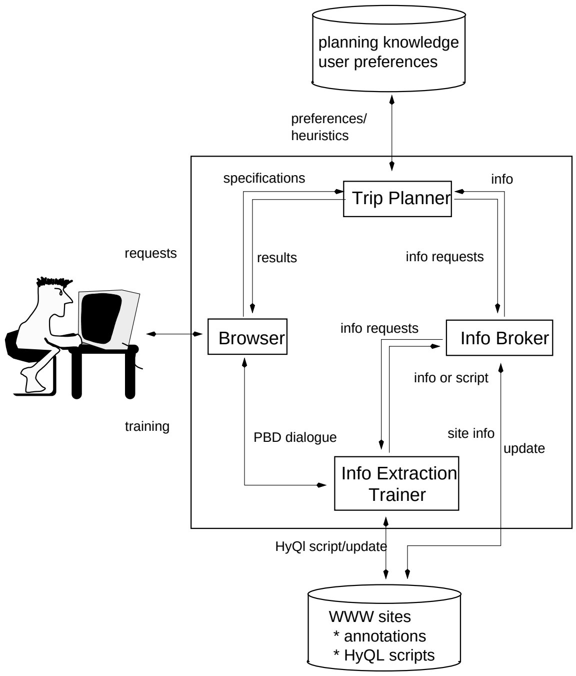

Figure 1: The TrIAs/ITA Architecture

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 question of how to extract the desired data—like availability of a hotel room—from HTML documents with possibly unknown structure.

The Architecture

In the following, we will introduce our agent architecture for a *Trainable Information Assistant* (TrIAs) which has been instantiated by an application plugin 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. There is no limitation whether the user is an individual person or a tour operator agent in a traditional travel agency.

The three core components of the architecture are the *application module* instantiated by the *trip planner* 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 as an information *consumer* and an information *provider* which is associated with the broker's current abilities. Regarding the consumer role the user specifies the desired trip using the specific interface (e.g. HTML-forms) provided by the trip planner. The planner is realized as a constrained-based process able to handle different means of transportation (flight and train schedules including fare tables and currency conversions, car rental services), able to integrate appropriate accommodations, etc. Using all these data it incrementally refines the trip specification considering the user's preferences.

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 locally not available—it sends appropriate information requests to the broker. This module maintains a rich domain model that basically contains descriptions of possibly heterogeneous information sources and mapping algorithms which realize a refinement from a high-level request to WWW-query expressions.[1] These are expressions in our SQL-style query language *HyQL* which provides among other things flexible concepts to support detailed WWW structure as well as document content access in a homogeneous and robust way. The information broker's interface to the Web is implemented by the interpreter for HyQL. If the currently relevant HyQL scripts can be successfully evaluated the broker hands over the results to the application module, thus satisfying the original information request.

But what if the broker is not able to extract the required information, e.g. because the layout of a web page has changed? In such a case the broker passes the relevant request to the information extraction trainer which starts a *programming by demonstration* (PBD) dialog with the user, that is the roles of information provider and consumer are exchanged. IET loads the relevant web pages to the browser[2] and starts a PBD-session by first communicating a problem description. The user either simply marks the required piece of information—e.g. the price of a hotel room—thus satisfying the *current* request or trains the system to perform this task autonomously in the future. The latter will be discussed in one of the subsequent sections. At the end of this *training phase*, the broker either obtains the missing information or the method to extract it from the given document. In any case the trip planner can continue its work.

[1] The broker's task is also to maintain statistics about connected and visited sites in order to adapt its heuristics for future information gathering.

The HyQL Query Language

There is a number of approaches concerned with WWW query languages, where each single approach has nice and useful features. But, none of these has satisfied all our requirements in the context of trainable information assistants. The language must support:

- detailed specification of WWW navigation and programmed search,
- detailed and flexible access to document structure and content,
- flexible referencing and selection scheme to reach robustness of queries against layout changes of documents,
- specification and use of user-defined abstractions (macros),
- homogeneous language definition fulfilling the needs of naive as well as expert users.

Our approach of a WWW query language called HyQL provides all of the desired features. Its main purpose is to operationalize information gathering processes, those which are already known and abstractly specified, but more interestingly those which are new and defined by the user in cooperative interaction with the system. E.g., the user can define a parameterizable abstract action of the kind *"select the partial city map for city X which covers the location of hotel Y"*. The abstract specification is then given by:

$$
\begin{aligned}
&\mathrm{hotel\_location}(+C, +N, -I)\\
&\quad \text{pre: } city(C)\\
&\qquad\quad hotel\_name(N)\\
&\quad \text{post: } street(hotel(C,N)) = S\\
&\qquad\quad \exists I : citymap(C,I)\\
&\qquad\qquad member(covers(S),\\
&\qquad\qquad\qquad features(image(I)))
\end{aligned}
$$

[2] The pages are non-visibly augmented with special features in order to support the PBD-process.

HyQL is a WWW query language, designed to implement the operationalization of actions of this kind. It is on one hand a means for the domain expert who specifies e.g. the information gathering tasks associated with a trip planner. On the other hand, it is the target language used in PBD-sessions to represent the intermediate and final hypotheses about the user's intended information selection concept.

In the following, we will discuss important features of HyQL supporting the realization of the aspects of information gathering just sketched.

HyQL considers the WWW as a computable dynamic graph structure where the nodes are static or dynamically generated documents and the edges are the links between them. The documents themselves are represented as HTML tree structures in a canonical form, which means that some obvious faults in documents are repaired and optional start or end-tags are introduced. HyQL is an SQL-style query language which provides a lot of functions for the exploration and selection of labeled graph and tree structures.
Now, we explain the most important concepts of HyQL.

Selecting Parts of Documents

The basis for the selection of document parts is the XML (xml 1997) linking and referencing concept operating on a labeled ordered HTML parse tree. Let T be such a parse tree, E a tag in T, i.e. a terminal subtree of T, and P the path from the root of T to E. There are many ways to characterize a specific element E in T: by specifying constraints on P, the content or other properties of E, by specifying a context of E and E's relative position to this context, etc. *Collections*[3] 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. There are collections of children, ancestors, descendants,... of a node. 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.
Example 1: The following sample expression is able to select the first four rows of the last table in the current document:

```
select interval
  first element TR in last element TABLE
    starting at root to
  fourth element TR starting here
```

A shorter form without syntactic cougar is given as:

```
select root,descendant(-1,table)(1,TR)..
       next(3,TR)
```

More complex selection operations can be defined by the `map` operation, which applies a selection expression S_1 on each item resulted by the evaluation of an expression S_2. The next example shows an application of `map` in order to select the second column of a table.
Example 2:

```
map second element TD
    on select all element TR
       in first element TABLE
```

Selection of Documents

HyQL provides different mechanisms for the exploration and selection of specific parts of the WWW. Three basic document quantification concepts are available.

- `select ... from document d such that d in URL`$_1$`,...,URL`$_n$
  The evaluation of this expression bounds that document to the variable `d` which is first completely fetched within a certain timeout[4]. The HyQL interpreter performs a parallel pull on the n documents characterized by their Url.
- `select ... from all document d such that d in URL`$_1$`,...,URL`$_n$
  The evaluation of this $\forall$-quantification demands that all documents URL$_i$, $i = 1, \ldots, n$ can be fetched within a certain timeout.
- `select ... from all reachable document d such that d in URL`$_1$`,...,URL`$_n$
  This is a weak form of the $\forall$-quantification, which fetches all documents reachable within the timeout.

A subgraph of the WWW to be explored and searched for a specific document can be specified by the use of path regular expressions. Therefore, we classify links as local (on the same server as the source), global, or empty (source equals target). Regular expressions built upon these categories can describe a more or less restrictive set of paths between the source and the potential target document. The search space associated with the path expressions can be further restricted by adding constraints on the nodes to be traversed. This can be extended to the concept of a dynamically *computed* path where the node to be reached next is a result of a function applied to the current document:

$$d \rightarrow f_1(d) \rightarrow f_2(f_1(d)) \rightarrow f_3(f_2(f_1(d))) \rightarrow \ldots$$

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

[4]If not set, a default value is taken.

here, d is the source document and $\rightarrow$ denotes a local link.

Example 3:

```
select content from document d3
such that document d in http://...  POST ...,
          d=>d2->d3,
          d2.url = select HREF in first A
                                in last TABLE from d
where match(d3.title,'News')
```

This script specifies a simple search procedure. `d` might be the document resulting by accessing the CGI-interface of a search engine which gets its current parameters using the `POST` method. Document `d2` is the first search result connected by a global link to `d`. In this document a local link is searched where the target document's title matches the keyword `'News'`. The search procedure could be used to find the News section of some company's homepage where only the name of the company is known.

Specifying and Using Macros

HyQL allows the definition of macros which are parameterized abstractions of HyQL-expressions. They can e.g. be used to specify new kinds of collections as given in the following example:

Example 4:

```
define element row_in_first_table($1)
as root,descendant(1,TABLE)($1,TR)

select all element row_in_first_table
from document d ...
```

Considering the PBD-scenario as described in more detail in the next section macros can be used to wrap user-defined annotations of document parts. Taking the table shown in Figure 3 as the current context the user could e.g. apply the above macro to characterize a new element type `message`. Successfully completing the training phase the system for example has additionally derived a macro defining a filter to select the relevant messages:

Example 5:

```
define element important($1)
as first element FONT with
         attribute SIZE=+2
         constraint substring_match($1)

select all element m message
from document d ...
where important('Telekom') applicable to m
```

The filter is here defined as a large-sized text portion which contains a substring matching a specific keyword. The application of the filter in the HyQL-expression above demonstrates another feature of HyQL: test or constraint predicates integrating structural properties of documents can be defined on the fly using the '`applicable to`' operator. A constraint '`E1 applicable to E2`' evaluates to true if the evaluation of the expression `E1` succeeds in the context of the value of `E2`.

Among others HyQL provides some primitives to produce output, i.e. to generate HTML and plain text, which let the HyQL-interpreter work as a CGI-program.

The Training Process

There exist at least two scenarios in which the user might want to teach the system how to extract relevant information from a web page. He either wants to create a new personal information service that integrates various sources from the WWW or the information broker came up with an unsatisfied information request from an application system like the the trip planner ITA. In the latter case the user is presented a short description of this request (e.g. "the price for a single room").

Before the training process starts, the HTML document currently under consideration is enriched by additional tags like `<WORD>` and `<SENTENCE>`—these are used to better characterize the user's navigation through the document text—as well as features extending the interactive functionality of the page.[5] 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 examples given then amounts to translating the corresponding subtrees into operational access procedures in terms of HyQL.

The rest of this section is organized as follows. The next section briefly characterizes the user's interaction with the system during a training session. Subsequently two learning tasks and their respective solutions are presented before further aspects like dealing with hyperlinks, applying recognizers, and defining macros are addressed.

The Interaction Style

The basic goal is to make the training process as painless as possible for the user. That is, the user should not be bothered with learning a programming language or even a protocol requiring some fixed interaction patterns. Instead, both trainer (user) and apprentice (info

[5] It is important to note that these changes remain invisible to the user, that is, the appearance of the document within his browser will remain the same.

broker) can take turns in initiating further iterations in a very natural kind of dialog in which the trainer

- provides examples (and possibly counterexamples) of the concept to be learned,
- gives additional hints facilitating its unique identification,
- asks the system to present its current hypothesis, and
- criticizes its performance.

The apprentice, on the other hand, can offer to

- identify the next positive example within the current document by itself,
- classify the user's selection using *recognizers*,[6] and
- present a simple natural-language translation of the HyQL script forming its current hypothesis.

As will become clear in the subsequent sections, the incorporation of user-defined macros enables the system to adapt this translation to the user's own categories.

The Learning Procedure

Two basic learning tasks influencing the underlying algorithms have to be distinguished. The concept to be learned can either describe a single piece of information that is contained exactly once in the document under consideration, or the user intends to teach the system how to extract a number of similarly structured pieces of information. An example of the first kind of *single-occurrence concepts* (SOCs) is the price of a single room given on a hotel web site. *Multiple-occurrence concepts* (MOCs) can be found wherever elements of a similar syntactic structure are enumerated, e.g. in a table.

As the algorithms used differ in both cases, the user can greatly simplify the learning process by initially indicating what type of concept (SOC or MOC) is to be learned.

Learning of SOCs Assume the user marks a certain portion of an HTML document, e.g. the second column of a <TABLE> as depicted in Figure 2. Note that the selected text is split in several non-cohering parts, the respective contents of the various <TD> tags.

[6] Recognizers are simple procedures that can identify pieces of information using their syntactic structure (like zip-codes or phone numbers). Their role will be discussed below.

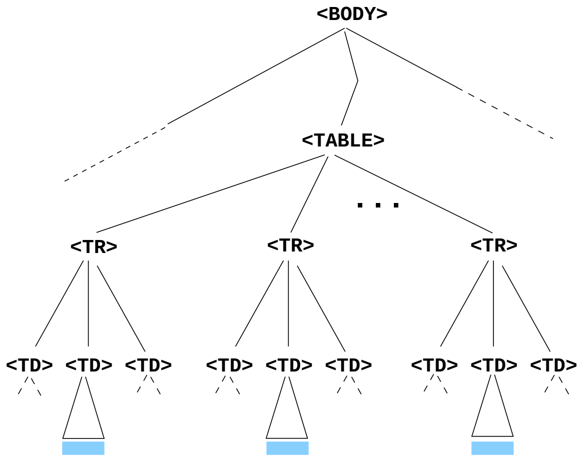

Figure 2: User's selection within the document parse tree.

The algorithm for the construction of a HyQL query then works in two phases. First the root nodes of "interesting" subtrees of the document parse tree are identified. A subtree is considered interesting if it covers parts of the user-selected text. The second step then consists of translating the paths leading to these nodes into HyQL statements.

Phase 1: Starting at the root of the parse tree, find the minimal number of subtrees completely covering the text selected by the user. For each of these subtrees the same search procedure is invoked. Iteration stops as soon as the complete leaf word[7] is contained in the current selection.

For the example of Figure 2, the first subtree found is the one originating at <TABLE>. While it completely covers the selected text, it also contains irrelevant portions (in this case the contents of the other table columns). Restricting the context to this tree the next iteration yields the subtrees corresponding to the various rows of the table (with root nodes annotated by <TR>). In the next round the <TD> subtrees are reached and the search stops.

Phase 2: A HyQL statement representing an operational description of extracting the selected text is generated by retracing the above search in reverse order. As an important substep the system tries to identify regularities in the resulting expressions that can be simplified using the `map` instruction (see Example 2).

In the above example, this is the case for the final step where the second <TD> element of each row has to be accessed. The complete HyQL expression derived is

[7] The leaf word of a tree is the string containing all its terminal nodes.

```
map second element TD on
select all element TR in
first element TABLE                         (1)
from document d in http://...
```

When learning an SOC it is often hard to decide how to describe the context, i.e. the syntactic features enabling the unique and robust identification of the relevant text portion even if the document itself was slightly modified. To this end, a number of heuristics were implemented that guide the learning process. If, for example, the selected text is found within a table, its column and row structure is used to address the selected area. In unstructured documents the system tries to detect graphical elements like horizontal lines ("... the second sentence after the first <HL>") or particularly formatted text (headers, bold-face words, ...) that can be used as "landmarks" for navigating within the document. The expressive power of HyQL including various addressing schemes—e.g. from the beginning or the end of a document—supports this search for discriminating features.

The user can additionally give hints by pointing at particular elements of the document (like images or lines) and tell the system to use them in constructing the context of the access script to be learned.

Remark: Note that the HyQL script (1) is very robust against modifications of the sample document as long as they do not directly affect the context (the *first* table within the document).

Learning of MOCs The task of characterizing an MOC using HyQL can be split in two subtasks.

- Find the *first* occurrence of the unknown concept in the document.
- Starting from there, find the *next* one.

Assume the document under consideration contains more than one table of the above-mentioned structure and the second column of each of them contains interesting information for the user.

In order to learn a unique characterization of MOCs the user has to identify at least two positive examples. After searching for corresponding subtrees in these examples, the system tries to find the first occurrence of the unknown concept in the given document. The user can facilitate this process by indicating that one of his examples actually is the first one.

Using this example an SOC expression like (1) is generated. Searching for regularities in the corresponding parse tree structures of all positive examples available then enables the system to find out which part of the context of this expression is crucial for switching from one occurrence to the next.

The HyQL statement for finding the second instance of the above example [8] is

```
map second element TD on
select all element TR in
first element TABLE next
from document d in http://...
```

By adding more and more `next` statements at this position, this script can be used to enumerate all instances of the user's concept.

Dealing with Hyperlinks

The examples considered so far only involved the extraction of relevant information from one single document. In many cases, however, information is dispersed on a number of web sites interconnected with hyperlinks. The news ticker from Figure 3 is a typical instance for this kind of information source. The upper frame displays an index file with the headings of various news items with a hyperlink to the corresponding text bodies contained in separate files.[9] Figure 4 displays the structure of such documents where D is the parse tree of the index file. The dashed lines represent hyperlinks to the associated documents D_1 through D_n.

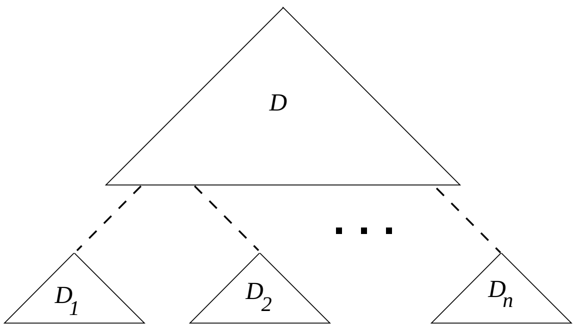

Figure 4: Parse trees of complex documents.

Treating hyperlinks as a special kind of parse tree edges the same mechanisms as described in the discussions of how to learn SOCs and MOCs can be applied. Note that HyQL provides all the operators required to navigate through sets of documents connected by hyperlinks.

The Integration of Recognizers

As already mentioned, recognizers are simple procedures that can classify text according to their syntactic

[8] That is, the second column of the next table to be found in this document.

[9] After clicking on the rectangle at the beginning of each line, the message text is displayed in the lower frame.

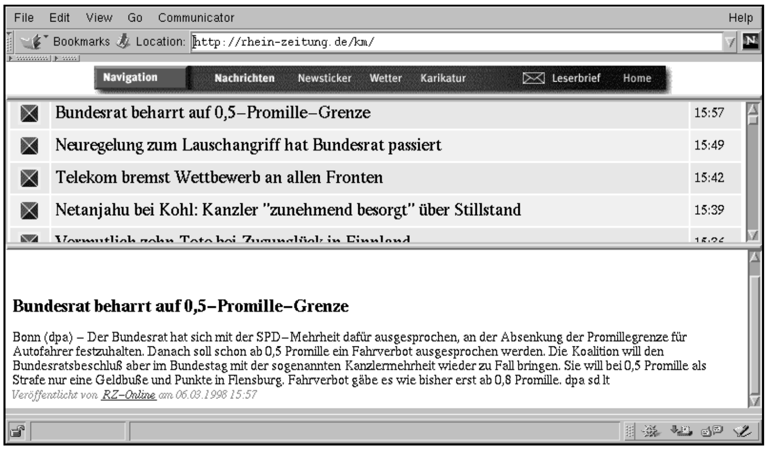

Figure 3: News ticker of a German press agency.

structure (cf. (Kushmerick, Weld, & Doorenbos 1997)). That is, given some string they can decide whether it contains a phone number, a zip code, a city name or whatever (depending on the application, domain-dependent recognizers have to be defined).

During the learning process they are used to characterize patterns within the selected text portions and produce more robust HyQL scripts by performing a kind of "type check" on the extracted pieces of information. Being able to guarantee certain properties of the data found is particularly important when other system components—like the travel planner ITA—use them as their input. In case of ambiguities—e.g. some German zip codes can be mixed up with telephone city codes—the system can ask the user to give the correct classification.

Assume the table column accessed by script (1) contains city codes. Integrating a corresponding type check yields the following expression

```
map second element x TD on
select all element TR in
first element TABLE
from document d in http://...
where city_code(x)
```

that filters out any table entry that does not form a valid city code (according to the test procedure `city_code`).

The Role of Macros

The HyQL language allows for the definition of macros. Within the context of training an information seeking agent, they can be used to

- efficiently synthesize more than one script for the same document and
- improve the communication between the system and its user.

If, for example, the user wants to extract several items from a block of text containing the address of a hotel, he can first train the system to reliably identify this portion of the document. Extraction of the more specific items like street name, zip code, and city name can then be addressed by first using the `address` script as a macro that determines the context of the scripts to be defined. Then the training phase continues as usual within this significantly restricted search space:

```
select second element x WORD in
first element ADDRESS
from document d in http://...
where city_code(x)
```

Besides accelerating the learning process, macros also serve the purpose to produce more compact HyQL scripts that can be easily communicated to the user using "his own words"—the user himself introduced `ADDRESS` as the name for this particular information category.

Depending on the user's expertise, the system can present its scripts in a straightforward natural language translation or in the actual HyQL code.

Additional Hints by the User

The user can support the identification of the selected text by pointing at specific text portions serving as "landmarks". Examples include words in titles or graphical items like horizontal lines.

Assume that in the above example the relevant text part is preceded by the phrase "Contact Information". Using this landmark for navigation within the document the learned HyQL script can look like

```
select second element x WORD in
first element ADDRESS
after first B matching 'Contact
Information'
from document d in http://...
where city_code(x)
```

Additional robustness can be gained by adding structural constraints (using the `applicable to` construct of HyQL) as was demonstrated in Example 5.

Related Work

With an increasing interest in building software agents that use information available on the Web as one of their main knowledge sources the problem of learning how to handle these sources at least semi-automatically arises. The *wrapper induction* method (Kushmerick, Weld, & Doorenbos 1997) aims at automatically constructing content extraction procedures for Web sources. The system inductively learns a wrapper generalizing from example query responses. Opposed to our approach the information sources considered (and also the wrapper class intended) are limited to have a very specific structure. Since their approach also requires more than a few examples to work, a user-interactive approach with online integration of new information sources like our TrIAs architecture is not adequately supported.

(Ashish & Knoblock 1997) aims at inducing a structure on the documents of interest by identifying potential headings, determining the hierarchical nesting of the various sections, and finally generating a parser that allows the extraction of structural units from the document. The user can support the system by pointing out mistakes.

In ILA (Perkowitz & Etzioni 1995) the *category translation* problem is concerned, i.e. how to translate information from Web sources into internal concepts of the information broker. We extend ILA's approach of using a fixed ontology in that we additionally allow the integration of a user-specific ontology.

The HyQL language incorporates some ideas from other approaches: both WebSQL (Arocena, Mendelzon, & Mihaila 1997) and W3QL (Konopnicki & 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.

Particularly the user interaction part of the PBD mechanism was inspired by work like (Maulsby 1994) where the requirements of a training dialog are described and various instantiations of instructible agents are presented.

Conclusion

This paper presented the TrIAS architecture for trainable information assistants. It described how a problem solver making use of information available on the WWW can be trained to extract useful data from HTML documents. Using programming by demonstration the user selects the relevant portions of a document and makes the system synthesize a HyQL procedure that enables it to autonomously repeat this selection for future tasks.

The expressiveness of the Web query language HyQL 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.

References

Arocena, G.; Mendelzon, A.; and Mihaila, G. 1997. Applications of a web query language. *6th Int. WWW Conference*. see also `www.cs.utoronto.ca/ websql`.

Ashish, N., and Knoblock, C. 1997. Semi-automatic Wrapper Generation for Internet Information Sources. *Proc. of the 2nd IFCIS Conference on Cooperative Information System.*

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

Kushmerick, N.; Weld, D.; and Doorenbos, R. 1997. Wrapper induction for information extraction. *IJCAI'97*, 729–735.

Maulsby, D. 1994. *Instructible Agents.* Ph.D. Dissertation, University of Calgary, Canada.

Perkowitz, M., and Etzioni, O. 1995. Category translation: Learning to understand information on the internet. *IJCAI'95*, 930–936.

Sqmlql: SGML Query Language. see `www.lpl.univ-aix.fr/projects/SgmlQL`.

XML: Extensible markup language Part 2. 1997. W3C Working Draft, `www.w3.org/TR/WD-xml-link`.