CHAPTER

Five

Trainable Information Agents for the Web

MATHIAS BAUER
DFKI

DIETMAR DENGLER
DFKI

GABRIELE PAUL
DFKI

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 chapter presents an approach to cooperative problem solving in which an information agent and its user try to support each other in achieving a particular goal. As a side effect the user can extend the agent's capabilities in a programming-by-example dialogue, thus enabling it to perform similar tasks autonomously in the future.

5.1 Introduction

Software agents are intended to perform certain tasks autonomously on behalf of their users. Examples include interface agents (Kozierok and Maes 1993), Web-browsing assistants (Lieberman 1995), and personal news agents (Billsus and Pazzani 1999), to name but a few. 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 is to identify precisely 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.

An ideal candidate for the role of such a supporting agent is a system user who 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. Consequently, it seems rational to ask her for help whenever the system gets into trouble. Programming by example (PBE) provides a feasible framework for the particular kind of dialogue required in such situations in which both user and agent use their individual capabilities not only to complement each other to overcome the current problem but also to extend the agent's skills, thus enabling him to deal successfully with a whole class of problems and avoiding similar difficulties—and thus additional training effort—in the future.[1]

Imagine a concrete application scenario in which a Web-based travel agent uses dynamic information located at various Web sites to configure a trip satisfying the user's preferences and constraints. Typical information

1. Throughout the rest of this article, we will refer to the user in the female form and the agent in the male form.

sources to be used in such a case include the Web sites of airlines, hotels, possibly weather servers, and so on. Unfortunately, many of these Web sites tend to change their look and structure quite frequently, thus exasperating agents who are not flexible enough to deal with this unexpected situation or at least recognize the fact that a problem exists at all.

Wouldn't it be good if this agent could tell his user about his problem and ask her to tell him what to do now and in similar situations occurring in the future? In the remaining sections, we will elaborate on this scenario and in particular describe the way the agent can ask for help without asking too much from the user—after all, the system is intended to provide some service to the user, not the other way round. So the role exchange between user and system (as service provider and consumer) should be as painless as possible to her. To this end, the agent should not remain passive and have the trainer do all the work but instead actively participate in the training dialogue and guide the teacher to give him just the right lessons to solve his problem. So, besides a new *application* for PBE techniques—the generation of scripts for the extraction of information from Web sites—we also advocate their use for a particular type of collaborative problem solving.

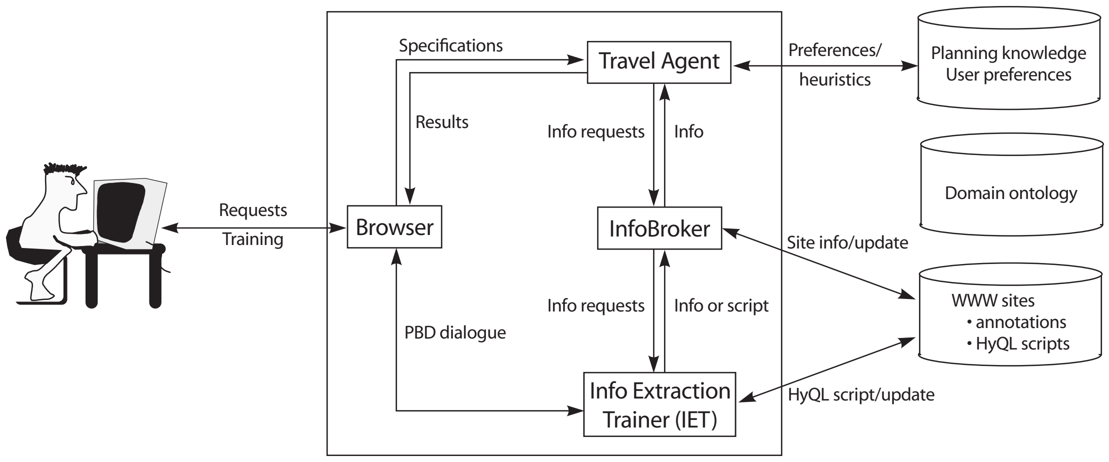
5.2 An Application Scenario

To illustrate both the kind of situation in which the aforementioned training dialogue takes place and the collaborative nature of such a session, consider the following instantiation of the trainable information assistant (TrIA) framework as depicted in Figure 5.1. Assume a user is preparing for a trip. Using her Web browser, she enters the relevant data, such as cities to be visited, budget limitations, and time constraints, leaving the rest to a Web-based travel agent who is expected to fill in all the missing details and suggest a journey satisfying the user's preferences. For most of this planning process, the agent has to make use of information that is not locally available but must be fetched from the Web at planning time. Examples include departure times from train or flight schedules, prices for hotel rooms, and so on.

Using the terminology defined by a domain ontology, the trip-planning agent formulates corresponding information requests to be answered by an *information broker* (called InfoBroker in the TrIA context). The latter has at his disposal a database of Web site descriptions consisting of

- the respective Web address (the URL);
- one or more query schemes;

FIGURE 5.1



A typical instantiation of the TrIAs scenario.

- the principal information categories to be found at that site, expressed in terms of ontological concepts;
- for each information category, a procedure (a HyQL script; see Section 5.3) implementing its identification and extraction.

The query schemes describe which information has to be provided to query a Web site and what can be extracted from the answer document. For example, a query to a flight server requires as minimum input the specification of departure and destination cities, and the preferred date for the flight. The returned information includes departure time, price, carrier, and so forth.

The InfoBroker combines these query schemes using a classical artificial intelligence (AI) planning approach and thus allows intermediate results from a variety of Web sites to be combined so as to best answer the original information request. (For a detailed description of this approach to query planning and information integration, see Bauer and Dengler 1999a.) If everything works out just fine (i.e., if all answers to the travel agent's questions can be found on the Web), the user is presented the final result in her browser.

The interesting case occurs whenever a relevant piece of information cannot be found at a particular Web site although it should have been there.

The typical reason for such a failure is the modification of the site's layout or structure. This often makes useless the information extraction procedure that was stored in the database characterization for this particular site.

The traditional approach of dealing with this kind of problem has at least two drawbacks:

- The user will be frustrated because the agent will not produce the desired outcome but instead an error message, at best (unless alternative sites that have *not* changed recently can provide the same information).
- The expert in charge of maintaining the InfoBroker's database has to program yet another procedure for dealing with the new look of this Web site.

To kill two birds with one stone, why not have the user assume a small part of the system maintenance (by updating the agent's knowledge base in an appropriate way) in exchange for a good answer (a suggestion for a trip), knowing—or at least hoping—that other users are doing the same, thus improving the overall system performance?

To be concrete, the problematic HTML document is handed over to the *information extraction trainer* (IET) along with the current information request (the agent's question whose answer was expected to be in this document but could not be found). After some preprocessing that remains hidden to the user, the document is opened in the user's browser. Some extra frames are used to guide the training dialogue in which the user initially simply marks the relevant portion to be extracted and possibly gives the system some hints on how to facilitate the identification of this particular piece of information. In the end, a new information extraction procedure is synthesized and inserted into the database for future use. To avoid frequent training sessions, the Web query language HyQL (Bauer and Dengler 1999b) is used as the target language. The next section will briefly describe HyQL and its use for information extraction before Section 5.4 gives a more detailed account of the actual training dialogue.

5.3 The HyQL Query Language

Our approach of a Web query language called HyQL is an SQL-like language that supports flexible selection of document parts as well as navigation through the Web. HyQL combines but also extends features found in related language designs with the aim to handle navigation as well as document processing on the level of both structure and content. The main purpose of HyQL is the operationalization of 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 flexible linking and referencing concept, we integrated features to be able to specify robust queries.

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 are specified by constraints on this structure and the contents of the nodes. Searching in the graph is supported by the specification of regular path expressions that, for example, allow specifying that only links on the originating Web site are followed until a depth of 3 is reached.

A sample task of navigation is filling a form on a Web page, submitting it, and reaching the result page. This process involves constraints between two document nodes named by their URLs, U_1 and U_2, respectively. The document of U_1 contains the HTML form that partially specifies U_2 (by the attribute-value pairs and action contained in the form). The pattern of U_2 is completed by an external interaction (by a user filling the form in her Web browser window or by a script setting the respective attributes). Now, establishing the edge between U_1 and U_2—that is, accessing the document of U_2—corresponds to submitting the form successfully in the browser.

The documents themselves are represented as parse tree structures in a canonical form, which means that some obvious faults in documents are repaired, optional start or end tags are added, and additional annotations such as *word* or *number* are integrated.

The expressiveness of HyQL allows document portions to be addressed in a variety of ways. The position of specific elements *E* in a document tree can be characterized by specifying constraints on the paths from the root to the relevant subtrees, constraints on properties of *E,* specifying a context of *E* and *E*'s relative position to it, and so forth. Collections play a key role in the specification of a position. A *collection* is an ordered set of related elements that results from a complete search over a subtree considering specific constraints on the search method used and the nodes to be considered according to their type and attributes. For example, the collection of all tables reached by traversing a document tree in a depth-first, left-to-right manner can be specified. All collections can be accessed as a complete list or indexed forward or backward. In addition to the single element and all collections, selections of intervals are also possible by specifying two single border elements (which may belong to unrelated collections) and implicitly all elements between them.

In the following, a sample HyQL script is presented that could be used, for example, as the program code of an appropriate CGI-Web access (the line numbers are *not* part of the script). It selects the pure table with the current weather forecast for the city of Berlin from the Yahoo! weather site.

```
1. select info T := root,descendant(1,table)
2. from document d1 such that
3. document d in http://weather.yahoo.com/Germany.html
4. document d ? document d1
5. d1.url = select root,descendant(1) href
6. from {select info A := root,descendant(1,a)
7. from document d
8. where A match “Berlin”}
9. where root,child(1,tr)(1,td)descendant(1,b) applies to T
10. matches “Today*”
```

Lines 1, 2, 9, and 10 specify an operation in the SQL style of selecting some part from a specific resource where some qualification conditions must be satisfied. These parts of the script specify that the first table from the collection of all tables (line 1) to be reached by a depth-first, left-to-right search (the keyword descendant) should be selected from the content of document d1 (line 2) that contains a bold-typed part in the first cell of its first row[2] whose content has the string `today` as a prefix (lines 9 and 10). Lines 3 to 8 contain constraints of how to access the content of document d1. Line 4 specifies that document d1 can be accessed by following a local link (remaining on the same Web site) originating in document d, whose URL is fixed in line 3 (i.e., the host part of the URL of d1 has been fixed). The URL of d1 (line 5) is further constrained by the fact that its document part is given by the href attribute of an anchor in document d whose representation matches with the word `Berlin`. In fact, document d contains links to a lot of German cities in order to access individual local weather forecasts.

Usually, a HyQL script is specified by a set of queries where `let` queries providing intermediate results precede the output-producing `select` queries. This supports the refinement process of information extraction on the syntactic level of the script by avoiding, for example, deeply nested `select` queries and allows a component-based configuration of scripts.

2. Consider the correspondences: cell - td, row - tr, bold-typed - b.

5.3.7 The Construction of Wrappers

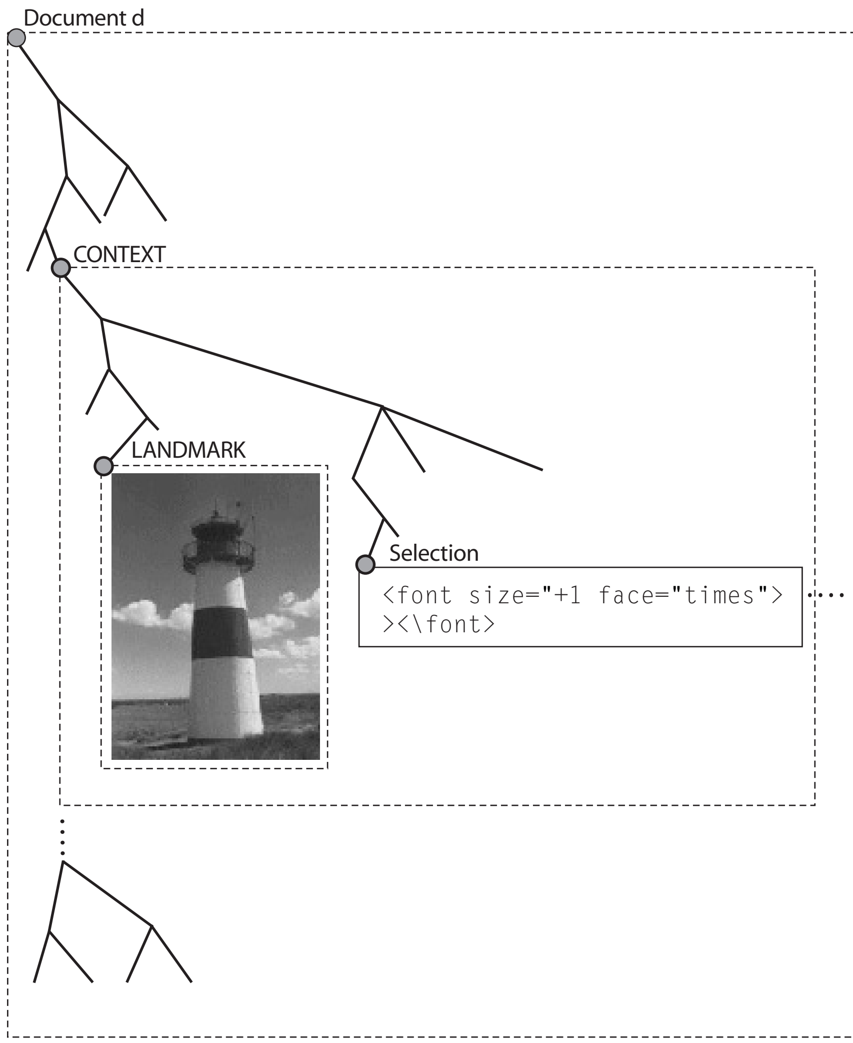
The TrIAs context requires the wrappers used for information extraction to be as robust as possible in the sense that they tolerate minor changes of the structure and layout of the documents they work on. As a consequence, it is necessary for the script not simply to implement a simple search, following strict paths in the documents' tree structures, but to exploit the document structure on a more abstract level.

The categories considered to be relevant with regard to the items to be selected are context, landmark, and characterization. A *context* is a part of a document covering and surrounding the selection that can be easily characterized and located (e.g., the only table contained in a document that has at least five columns and more than three rows). A *landmark* is a prominent feature of the document that is in close relation to the selection and works as a navigational hint (e.g., a particular image just before the text to be selected). A *characterization* of a selection deals with the specific format and style of the selected items. HyQL scripts can be constructed in such a way as to mirror the use of the structural criteria just mentioned. The following skeleton of a script explains the idea.

```
{ let info CONTEXT := . . .
from document d such that . . . }
{ let info LANDMARK := . . .
from CONTEXT
where . . . applicable to LANDMARK }
{ select here,following(1,font,{size = "+1"})
from LANDMARK in context CONTEXT }
```

The first part of the script assigns the variable CONTEXT to some specific part of the document named d (i.e., the result of some selection operation is now internally available for further processing). The second part of the script tries to find a specific landmark in the document portion assigned to CONTEXT. If it can be located, then it is assigned to the variable LANDMARK and again is internally available. Now, the last part of the script specifies that we start a search from the LANDMARK location in the document to find a font tag with a specific size attribute, but the search is limited to the area covered by CONTEXT. If this font is found, then it is provided as an output of the script. More complex scripts can be built, for example, by combining more than one of the script blocks of the kind sketched earlier.

FIGURE **5.2**



Extraction refinement process of a HyQL script.

Figure 5.2 explains the idea of the script illustrated here from the document tree point of view. HyQL parses each relevant HTML document and transforms it into a tree representation where the nodes correspond to HTML tags and the outgoing edges represent the content and children tags covered by these tags. Then, the context part of the respective HyQL script is given by the subtree of the document tree labeled by `CONTEXT`. This subtree contains at a specific location a landmark (e.g., a salient feature such as a specific image)—that is, a subtree of a specific kind labeled as `LANDMARK`. The document portion that should be selected as the relevant information can be located considering a particular relation with regard to the landmark position (e.g., the first specific font tag that follows the specific image object[3]).

5.4 The Training Dialog

Throughout the rest of this section the terms *user* and *trainer* will be used synonymously (and referred to in the female form). *Learning agent* (or simply *agent,* addressed in the male form) refers to the InfoBroker who is to be trained on how to extract a particular piece of information from an online document. The IET (see Fig. 5.1) provides the interface in which both partners can collaborate to accomplish this common goal.

As mentioned in Section 5.2, the training dialogue is invoked whenever the InfoBroker does not succeed in extracting some piece of information (e.g., the price for a hotel room) from an online document delivered by some Web server. In this case, the IET

1. repairs the HTML code of the document under consideration (if necessary),
2. enhances the so obtained document with additional tags (HTML formatting instructions) and JavaScript code, and
3. loads it into a new browser window containing additional buttons for PBD functionality (see Fig. 5.4 later) and explains the user the current problem.

These points have to be clarified. First, regarding point 1, most documents found in the Web are not made of absolutely correct HTML code. Instead, most authors rely on the various browsers' capabilities to somehow produce a satisfying rendering resembling the author's intended design. As HyQL uses the parse tree of these documents as its main data structure, it is

3. "Follows" in the sense of traversing the subtree `CONTEXT` in a depth-first, left-to-right manner starting at the node `LANDMARK`.

recommendable to bring them into a canonical form before starting to produce a wrapper.

Regarding point 2, the enhanced document structure is used in two ways. First, the JavaScript code serves the purpose to add some interactive functionality to the document such as highlighting some portion of the document by moving the mouse over it. Additional tags give the whole document a more fine-grained structure, allowing even single words or special characters to be recognized and providing a unique identifier for each document part. As a result, it is easy to find out exactly which document portion was selected (highlighted) by the user. Note that these first steps remain invisible to the user; that is, the visual appearance of the document in her browser will remain unaffected.

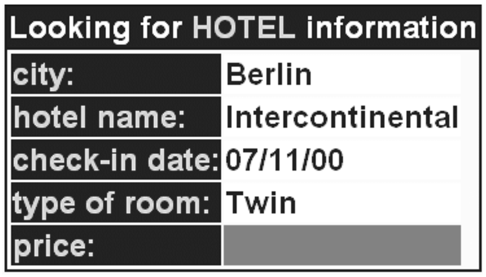
Finally, regarding part 3, the training dialogue is initiated as a reaction to the InfoBroker's futile attempt to extract information from a particular document in order to satisfy an information request forwarded by the travel agent. The agent uses all the information found so far in order to explain the current task to the user. In the example described in Section 5.4.4 the agent tries to find the price for a twin room in some Berlin hotel. Figure 5.3 shows how the user is presented this problem by referring to the already known values for hotel name, city, and so forth.

The actual training dialogue (Figure 5.4) itself starts with the user accepting the task presented by the agent and selecting that part of the document containing the desired information—in this case, the price of "475.00". Then the following cycle starts:

1. The agent synthesizes a wrapper using only the information he obtained so far and computes a numerical measure of goodness for it.
2. He checks what could possibly be done to improve this wrapper and suggests the corresponding actions to the user (examples include the definition of a landmark or a context; see Section 5.3.1 for details on the general structure of wrappers); if no improvement can be expected, the agent suggests to terminate the training dialogue.
3. The user picks one of the actions suggested by the agent and executes it with the agent's aid.
4. The process returns to step 1.

These steps will be explained in some detail in the following sections.

FIGURE 5.3



Looking for HOTEL information	
city:	Berlin
hotel name:	Intercontinental
check-in date:	07/11/00
type of room:	Twin
price:	▨

Explaining the current task.

5.4.1 Wrapper Generation and Assessment

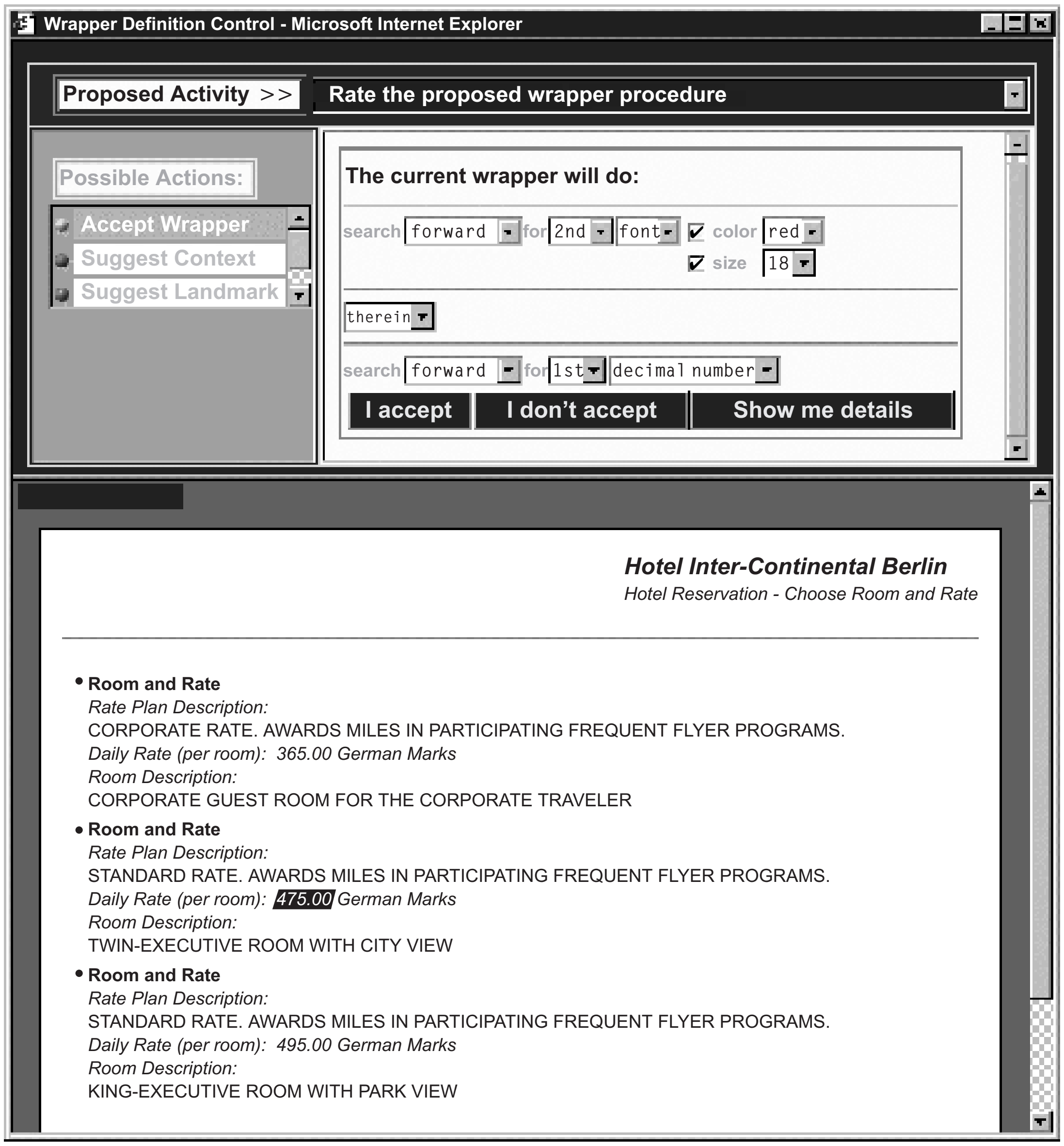
Given some selected portion of a document, the agent computes the assessments of possible wrappers, disregarding cases under a certain threshold. To this end, he uses a hierarchy of wrapper classes for the characterization of the selection. Each class is a kind of template containing wrapper building blocks (implemented as HyQL snippets) with parameters yet to be specified and is associated a numerical valuation reflecting its estimated utility. This measure mainly refers to the expected robustness of a wrapper containing a HyQL construct of some sort.[4] For example, those exploiting the document structure can be expected to be more robust than those simply counting the distance of some selection from the beginning of the document.

Apart from this intrinsic valuation of the various wrapper classes, a wrapper assessment must also take into account the cost for localizing the current selection, given navigational aids such as contexts, landmarks, and a characterization of the target selection on the basis of syntactic features such as style and font.[5] Here the basic idea is that the more complicated this navigation is, the more likely it is to fail due to document modifications. For example, always finding the *first* occurrence of some text written in red

4. Whenever a wrapper fails to produce the desired information and a training dialogue becomes necessary, it is analyzed and the assessment of the components responsible for this failure is decreased.

5. Referring to Figure 5.2, this corresponds to moving from one localization point (e.g., the start of a context) to the next one—in this case, the landmark—until the selection is reached.

FIGURE 5.4



Proposed Activity >> Rate the proposed wrapper procedure

The current wrapper will do:

search forward for 2nd font ✓ color red ✓ size 18

therein

search forward for 1st decimal number

Hotel Inter-Continental Berlin
Hotel Reservation - Choose Room and Rate

- **Room and Rate**
  Rate Plan Description:
  CORPORATE RATE. AWARDS MILES IN PARTICIPATING FREQUENT FLYER PROGRAMS.
  Daily Rate (per room): 365.00 German Marks
  Room Description:
  CORPORATE GUEST ROOM FOR THE CORPORATE TRAVELER
- **Room and Rate**
  Rate Plan Description:
  STANDARD RATE. AWARDS MILES IN PARTICIPATING FREQUENT FLYER PROGRAMS.
  Daily Rate (per room): 475.00 German Marks
  Room Description:
  TWIN-EXECUTIVE ROOM WITH CITY VIEW
- **Room and Rate**
  Rate Plan Description:
  STANDARD RATE. AWARDS MILES IN PARTICIPATING FREQUENT FLYER PROGRAMS.
  Daily Rate (per room): 495.00 German Marks
  Room Description:
  KING-EXECUTIVE ROOM WITH PARK VIEW

A sample training dialogue.

within some section of a document can be expected to be more robust than the search for the twenty-third occurrence (due to the fact that the chance for something unforeseen happening in between is much higher).

To summarize, at each step the agent must consider all reasonable combinations of landmark, context, and selection characterization. Their respective assessments are computed taking into account

- the valuation of the selection characterization and its localization cost with respect to the other defined parts (i.e., the estimated robustness of the wrapper and the actual localization cost);
- the valuation of context and/or landmark and the resulting localizations;
- the user's acceptance, depending the user's expertise. Components explicitly suggested by the user obtain a higher value than those proposed by the agent and simply accepted by the user (disussed later).

5.4.2 Suggesting an Action

The agent has at its disposal a task *library* containing "recipes" of how to construct a wrapper of a certain class. The actions they are made of include

- characterizing the selection,
- defining a landmark,
- defining a context, and
- accepting the current wrapper.

More details about these actions will be given in Section 5.4.3.

To suggest an appropriate action to be carried out next, the agent

- evaluates the document structure to identify those wrapper classes that are, at least in principle, applicable to the current document;
- checks what remains to be done to complete the corresponding task;
- computes the *expected utility* of each feasible action; and
- presents them to the user in a ranked order reflecting their respective estimated usefulness, including a brief help text describing the meaning of

these actions. Note that the user is free to choose *any* feasible action, not only the one considered best by the agent.

How can the expected utility of an action be estimated? To make the whole training process as painless as possible to the user, only those actions should be suggested that actually advance the dialogue by providing valuable information to the agent. The numerical wrapper assessment explained in the previous section forms a good basis for estimating the effect of an action on the expected wrapper quality. Using the formula

$$EU(a, u, n) = [ass(w_a) - ass(w_{curr})] \cdot P_u(a) - annoy\,(u, n), \tag{1}$$

the expected utility for action *a* and user *u* can be computed. Note that *n* is the number of actions already carried out during the current training dialogue; $ass(w_a)$ and $ass(w_{curr})$ are the assessments of the best wrapper possible *after* and *before* executing action *a,* respectively. $P_u(a)$ is the probability of user *u* successfully carrying out action *a,* and *annoy* is a function that, depending on the user's characteristics, grows monotonically as the length of the training dialogue increases. Formula (1) thus represents the expected increase in wrapper quality decreased by a penalty for annoying the user with yet another action.

In case none of the available actions is assigned a positive utility value—that is, if no action is expected to provide a sufficient improvement to justify continuing the training process—the agent suggests to finish the dialogue.

This is the case in the situation depicted in Figure 5.4. As the upper left window titled Possible Actions indicates, Accept Wrapper (the final action of each dialogue) is preferred over Suggest Context and Suggest Landmark. The details of how the user's preferred action is actually being executed are explained in the following section.

5.4.3 Executing an Action

Once the user has selected an action to be executed from the ranked list of alternatives *other than* terminating the dialogue, the agent tries to give optimal support to facilitate the user's task. (This special case will be described later.) To this end, he applies a number of strong heuristics to identify and suggest candidates for the wrapper component the user is just trying to define.

To find a candidate for a landmark such as a heading, the agent looks for boldface text (like **Room Rate** in the example of Figure 5.4), text followed by

S
R
L

a colon, or an image. Tables and regions enclosed between two objects (e.g., horizontal rules) are considered to be good candidates for contexts. Any syntactic feature that distinguishes the selected text from its surroundings (e.g., a special color or font) could make a good characterization.

With each possible component found, the agent generates a wrapper, computes its quality, and from these results derives a ranked list of suggestions for contexts, landmarks, and so forth. By clicking through this list of suggestions (and thus highlighting the corresponding parts of the document), the user can explore the set of alternatives provided by the agent and simply accept one of these suggestions or define a new one on her own. In the case of landmarks and contexts, the latter can be easily accomplished by just marking the corresponding region with the mouse. The agent then simply checks for structural correctness and, for example, rejects contexts that do not contain the target selection. When it comes to characterizing a selected area, the agent lists all applicable syntactic features (e.g., font, color, size etc., but also possible "types" such as decimal number or ZIP code that can be recognized automatically) from which she can select the ones she considers relevant. So in any case, the user is forced to make only structurally correct decisions. This interplay between the complementary capabilities of agent and user will be discussed in Section 5.6.

As already mentioned, terminating the training dialogue is a special case. Whenever the user decides to accept the agent's currently best wrapper, she is presented with a simple representation of this wrapper in terms of its components and the way they interact with each other (see Fig. 5.4). The user can easily modify this wrapper by switching components on and off and changing the way they are combined. Every time, the effect of such a modification is immediately displayed in the lower window containing the current HTML document. This provides a way for the user to experiment with the agent's construction and find out whether it actually does what she intended.

5.4.4 Example

Take as an example scenario for PBE the document shown in Figure 5.4. We assume the user is planning a trip to Berlin with a stay at Hotel Inter-Continental. With the wrappers at hand, the trip planner has not been able to extract the room rates from the hotel page, so the agent initiates a dialogue with the user.

The PBD dialogue is presented in the browser together with the relevant document in the lower frame. The upper frame is reserved for the communication with the user. The *Proposed Activity* field verbalizes the next step the user should take for a successful wrapper construction. An interaction history with undo mechanism is available through a pull-down menu in the top line. The list of ranked possible actions is given on the left side of the upper frame. A green light indicates that the corresponding action is explicitly suggested by the agent, whereas actions with a red light can be executed "at the user's own risk" as they will not lead to the best possible wrapper according to the agent's computations.

The best choice in the agent's view is already preselected. Most actions require an additional communication with the user. This is done in the area to the right where the user is presented the respective submenus and relevant information for her decision making (e.g., wrapper components proposed by the agent). A red box around an area characterizes a currently relevant interface part for a better orientation of the user.

In an initial step, the user is presented with the task at hand (see Fig. 5.3), which in this case is the identification of the price for a twin room. The first—and only—action applicable is the selection of the desired information. The user marks the price of "475.00" displayed in a red font, as are all other prices on the Web page.

Having done so, the proposed actions comprise as a first choice to further characterize the selection, which the user accepts. The agent lists the applicable syntactic features (red color, size 18, and italic font, from which the user selects the first two) and types (in this case, only "decimal number," which is also accepted by the user).

Applying his aforementioned heuristics, the agent tries to identify reasonable contexts and landmarks he could suggest to the user to improve the current wrapper. However, due to the limited capabilities of these heuristics, he fails to find any document features that could be used as additional aids and make the wrapper more robust. Consequently, he suggests that the user accept the current wrapper, thus terminating the training dialogue.

After accepting this suggestion, the user is presented with the simple representation of the current wrapper displayed in Figure 5.4 and asked to check whether this is what she wanted. As already mentioned, the user can test what effect minor modifications of the various wrapper components and their combination will have on the search result. The current wrapper exclusively makes use of the syntactic features of the selected text area (looking for the first decimal number in the second occurrence of a text of size 18 in red) and thus ignores the user's reason for selecting just *this* occurrence of a room price in this document instead of the first or third one—namely, the fact that only this price applies to a twin room.

FIGURE 5.5

```
{ let info X1 := root, descendant(1,font,{color=”red”
    size=”18”})
    from document d in ...
    where text root,ancestor(1,li)
                   child(-1,br)next(all,span
          applies to X1 matches “TWIN EXECUTIVE ROOM*” }
{ select info X2 := root,descendant(1,span)
    from X1
    where X2 recognized as ‘decimal_number’ }
```

The final version of the wrapper.

She decides not to accept this wrapper and adds a landmark to it to connect more strongly the wrapper’s functionality to the *semantics* of the document. Using his purely syntax-based heuristics, the agent suggests a number of landmark candidates, none of which is suited to robustly identifying the price of a twin room. In the given example, the agent suggests using one of the occurrences of **“Room and Rate”** or the second occurrence of *“Daily Rate (per room):”.* The user, recognizing the semantic relationship between the words “TWIN EXECUTIVE ROOM” in the room description and the price for a *twin* room suggests using these three words as a navigational aid. After integrating this landmark into the wrapper (and giving this component an especially high valuation as it was suggested by the user herself), the agent again checks whether any further improvements are possible. As this is not the case, the user is again advised to accept the current wrapper and terminate the training dialogue, which she does.

Figure 5.5 depicts the final wrapper generated by the InfoBroker and used to satisfy the travel agent’s information request.

5.5 Lessons Learned

Although we have not yet performed a rigorous evaluation of the dialogue strategies sketched here, a few general lessons can already be derived from the first prototype of the PBD environment (Bauer and Dengler 1999a). This first version did not provide ranked suggestions for future user actions but came with a simple graphical interface that left all decisions exclusively to the user. In particular, the following two problems repeatedly occurred in our own interaction with the system:

What to do next? To make this decision, the user has to have some idea of what benefit the learning agent will have from some action taken by the user (i.e., which action would provide the most valuable information for the agent).
Can I stop? Related to the first point is the question of whether or not the user should continue the training process at all. After all, it makes no sense to provide more and more information to the learning agent if he has already acquired enough knowledge about the task at hand to generate a good solution. To make this decision, the user again has to reason about the potential impact of further actions on the quality of the learning result.

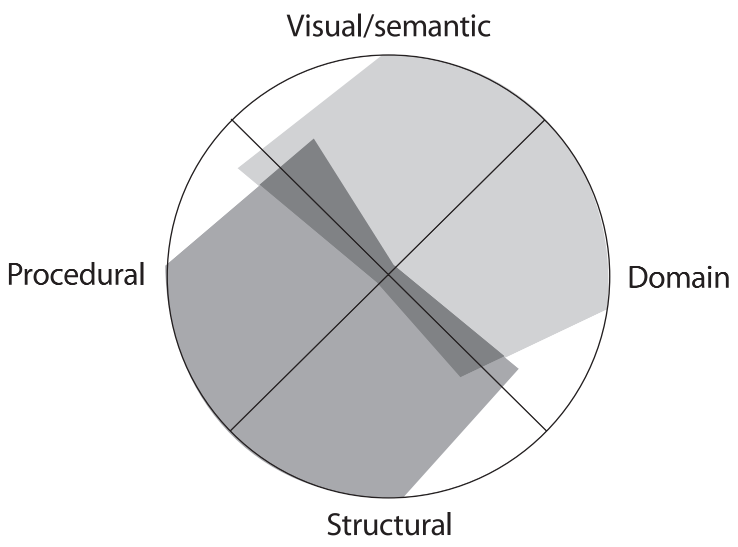
To overcome these difficulties, we analyzed the training situation at the respective knowledge levels of the two partners involved. The results of this analysis led to the utility-based approach to dialogue guidance described earlier, and they will be presented in some detail in the next section.

5.6 The Communication Problem

Effective communication between user (trainer) and agent (student) is hard to achieve. This is particularly true in scenarios such as the one sketched in Section 5.2 in which an agent is to be trained to interact with an already existing application that was not designed to be “programmed” this way. One of the most fundamental obstacles is the lack of shared knowledge or insight into the various aspects of the training task at hand. As depicted in Figure 5.6, at least four different types of information play a key role during a training dialogue:

- *Structural* knowledge refers to the internal properties of the programming domain (in this case, the HTML structure of a document to be processed).
- *Procedural* knowledge refers to the understanding of (at least) the basic concepts of the target programming language (in this case, HyQL) and the way the programs to be developed are intended to work (here, the principal functioning of wrappers).
- The *visual/semantic* category is composed of the optical perception of the application system to be dealt with, the representation of domain

FIGURE **5.6**



Knowledge shared by user and agent.

objects, and its interpretation allowing, for example, semantic relationships to be derived. Examples include the rendering of an HTML document in a browser and the identification of some particularly formatted portion of text as a heading describing the subsequent paragraph.

- Finally, *domain* knowledge includes the basic understanding of concepts from the application domain (e.g., about hotels, room categories, etc.) and the wording typically used to describe them.

These various categories are not mutually independent. Rather, a lack of structural knowledge prevents a sufficiently deep insight into the procedural aspects of the task at hand; little or no domain knowledge exacerbates the interpretation of visual encoding of information beyond purely syntactical aspects. Unfortunately, only a relatively small part of this information is shared by the user and agent and can thus be used for communication purposes.

As depicted in Figure 5.6, the learning agent, for example, possesses deep insight into the (programming) domain *structure*. In the given example, this refers to the system's ability to evaluate and exploit the underlying document structure induced by the HTML code.

While the typical user can be expected to have some basic understanding of HTML at best, this overlap usually proved to be insufficient to explain the system behavior—its suggestions regarding next actions or wrapper components—referring to such structural document properties.[6]

The user's most prominent capability is in interpreting and understanding the *visual appearance* of a document in her Web browser. Doing so, she can easily identify "important," never-changing aspects of a document and *semantic* relationships among objects (e.g., a graphic and its title) by exploiting her background domain knowledge. The system, on the other hand, has to rely on a number of more or less feasible heuristics (e.g., "the first line of a table column typically contains a header describing its contents") to at least make a guess of which objects are related and might thus make a good navigational aid.

The coverage of *procedural* aspects of the training task (i.e., details of the target language and the general functioning of wrappers) largely depends on the user's experience in such training activities, on the one hand, and on the system's learning bias, on the other hand.

Even without taking into account misconceptions on either side,[7] this discussion indicates that an effective communication providing perfect mutual understanding of both partners is almost impossible.

What are the consequences for the training dialogue in the TrIAs scenario? Obviously there exist two almost disjoint, complementary *competence areas.* The learning agent suggests the next actions to be taken or wrapper components (e.g., a landmark) to be used mainly based on its understanding of the underlying HTML structure. The user—in her role as a trainer—evaluates these suggestions based on her domain knowledge and the visual impression of the the document's rendering in the browser. Depending on the user's estimated expertise, either the agent autonomously decides on how to continue the training dialogue—that is, the user is only presented the seemingly best action at each point in time—or the user is free to accept or ignore the agent's suggestions, taking over control by herself.

---

6. The fact that the same visual rendering can be achieved using a number of different HTML encodings (just think of the many creative ways to use tables) (aggravates this problem, because it is almost impossible to guess correctly the actually used HTML code just by looking at its rendering.
7. Some of the heuristics used by the system might be perfectly wrong, as might be a user's guess of the document structure derived from its visual appearance.

Adding image-processing capabilities to the learning agent, as is done in chapter 19 of this book, enables it to (at least partially) understand the structure and functionality hidden in the user interface. In terms of the previously mentioned categories, this means that the agent's coverage of the "visual" information portion is significantly extended. Equipped with this enhanced insight into the visual appearance of an application, the agent can try immediately to interpret and generalize the user's interaction with the various interface elements, thus enabling it to program applications without an API.

Applications specifically *designed* to be programmed by example try to bridge the gap between user and system by providing a visual access to both the programming domain structure and the procedural aspects of the program to be developed. A typical example of this class is Stagecast (see chap. 1), a system intended to enable its user to program a kind of video game. The world inhabited by various kinds of creatures has the structure of a (visually perceivable) grid, and program steps consist of simple state transition rules represented by the respective states before and after rule application.

Even here, however, not all interaction can be solely grounded on visual perception. Instead, the user must be willing to delve into some of the more advanced features of the system to describe what is called "hidden states" in McDaniel (2000). This notion refers to additional aspects of the world that cannot be inferred from just one example but have to be explicitly stated by the user or inferred by the system. In the Stagecast example, such additional information includes abstractions such as "*any* kind of object should be in this place in order to make the rule applicable" or "variables" that represent the internal state of the characters.

This perspective is somewhat similar to the one taken in the Gamut project (see chap. 8) in which the user is considered to be actually programming. That is, the user is in charge of conveying all required information to the system, which in turn has to provide appropriate communication channels to facilitate this task for the user.

Again, this approach differs from the TrIAs perspective, in which the user initially did not intend to *program* a system but to *use* it. Consequently, she cannot be expected to be infinitely patient and willing to invest her time and effort to teach the system something she expected it to already know. In other words, it's not (only) about making *learning* as easy as possible for the agent (compare VanLehn 1987). Thus, the system's contribution must clearly exceed the simple check for structural properties of a document. In TrIAs this additional accomplishment is achieved by the library of numerically assessed wrapper components and the strong heuristics enabling the system to hypothesize semantic relationships among (graphical) objects.

5.7 Another Application Scenario

One peculiarity of the TrIAs scenario was the exchange of the roles as service provider and consumer between user and system. This made it necessary to consider carefully the user's "felicity" (in contrast to concentrating exclusively on the learner's state as described in VanLehn 1987).

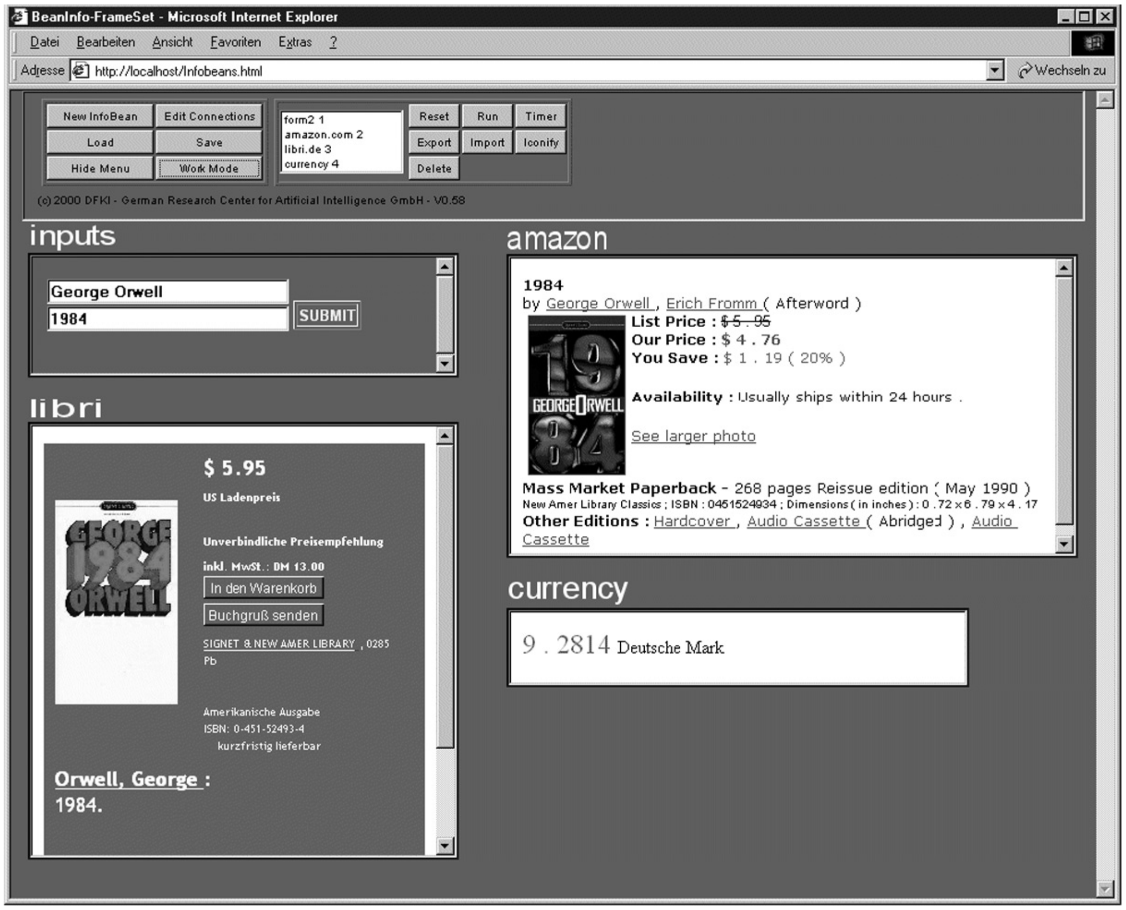
However, not only applications in which a (possibly unwilling) user happens to be involved in a training session benefit from the careful design of their trainable components. In fact, the same PBE approach as described earlier was used to implement the InfoBeans system (Bauer, Dengler, and Paul 2000) in which even naive users could configure their own Web-based information services, satisfying their individual information needs.

Figure 5.7 depicts an *InfoBox,* a personal information system composed from a number of simple information services. Each of these so-called *InfoBeans* represents an information agent that deals with one particular information source only. Using HyQL wrappers, these agents can query Web servers, extract interesting pieces of information from the documents delivered, and communicate these through input and output channels to the other InfoBeans, thus providing the basis for effective information integration. Whenever the user wants to create a new InfoBean or modify or extend an existing one, a training dialogue similar to the one described earlier is initiated.

The sample InfoBox depicted in Figure 5.7 compares the prices of two online bookstores. To this end the initial input, author and title of a book, are forwarded to the InfoBean dealing with Amazon.com, from which the book price in U.S. dollars is extracted and forwarded to an online currency converter to obtain the value in German marks. Additionally, the corresponding ISBN number is extracted and delivered to the InfoBean in the lower left corner that addresses Libri.de and tries to find the same book. This way a comparison shopping system, although admittedly simplistic, can be implemented.

Because the InfoBeans application is in the tradition of PBD systems (e.g., Lieberamn, Nardi, and Wright 1999) in which the user trains the system to recognize certain types of situation to be dealt with autonomously, we removed the "annoyance" factor when assessing the expected utility of

FIGURE 5.7



http://localhost/Infobeans.html

form2 1
amazon.com 2
libri.de 3
currency 4

(c) 2000 DFKI - German Research Center for Artificial Intelligence GmbH - V0.58

George Orwell
1984

1984
by George Orwell , Erich Fromm (Afterword)
List Price : ~~$ 5 . 95~~
Our Price : $ 4 . 76
You Save : $ 1 . 19 (20%)

Availability : Usually ships within 24 hours .

See larger photo

Mass Market Paperback - 268 pages Reissue edition (May 1990)
New Amer Library Classics ; ISBN : 0451524934 ; Dimensions (in inches) : 0 . 72 x 6 . 79 x 4 . 17
Other Editions : Hardcover , Audio Cassette (Abridged) , Audio Cassette

$ 5.95
US Ladenpreis

Unverbindliche Preisempfehlung
inkl. MwSt.: DM 13.00

SIGNET & NEW AMER LIBRARY , 0285
Pb

Amerikanische Ausgabe
ISBN: 0-451-52493-4
kurzfristig lieferbar

Orwell, George :
1984.

9 . 2814 Deutsche Mark

A sample InfoBox.

some system suggestion. Experiments will show which of these versions will have the better user acceptance.

5.8 Related Work (Non-PBD)

Instantiations of TrIAs are in the tradition of information integration systems such as the Information Manifold (IM) and Ariadne (see Levy, Rajaraman, and Ordille 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—for example, 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*) 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 such as ILA (Perkowitz and Etzioni 1995).

Current wrapper generation approaches such as those in Hsu (1998) and Muslea, Minton, and Knoblock (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, Weld, and Doorenbos (1997), Hsu (1998), and Muslea, Minton, and Knoblock (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.

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 semiautomatically arises. The wrapper induction method (Kushmerick, Weld, and 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.

In ILA (Perkowitz and Etzioni 1995), the *category translation* problem is concerned—that is, 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, and Mihaila 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 (1997) is a nice SQL-style programming language for manipulating SGML (HTML) documents but lacks any navigation constructs. XML (“Extensible Markup Language-1997) allows with its linking concept based on XML parse trees a precise specification of resource locations, especially within documents.

5.9 Conclusion

We have tried to make a case for the use of PBE techniques not only in an initial, offline training phase but also during the execution of the procedures so acquired. As was mentioned, this particular setting requires the user to be taken into account much more carefully. After all, she has to “repair” the faulty behavior of a system or agent from which she expected some useful service.

Early informal tests with nonexpert users indicate that the training mechanism provided enables (many) end users to deal successfully with delicate problems of identifying and extracting information from Web-based information sources. Besides the two application scenarios sketched earlier—the TrIAs framework that can be instantiated with a number of different applications and the InfoBeans system—many other uses of instructable information agents are conceivable, ranging from intelligent notification services to data warehouses.

References

Arocena, G., A. Mendelzon, and G. Mihaila. 1997. Applications of a Web query language. In *Proceedings of the 6th International WWW Conference* (Santa Clara, Calif.); www.cs.utoronto.ca/~websql/.

Bauer, M., and D. Dengler. 1999a. InfoBeans—Configuration of personalized information services. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI ’99)* (Los Angeles), ed. M. Maybury. New York: ACM Press.

———. 1999b. TrIAs: Trainable information assistants for cooperative problem solving. In *Proceedings of the 1999 International Conference on Autonomous Agents (Agents’99)* (Seattle, Wash.), ed. O. Etzioni and J. Müller. New York: ACM Press.

S
R
L

Bauer, M., D. Dengler, and G. Paul. Instructible agents for Web mining. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI 2k)* (New Orleans), ed. H. Lieberman.

Billsus, D., and M. Pazzani. 1999. A hybrid user model for news story classification. In *User modeling: Proceedings of the Seventh International Conference* (Banff, Canada), ed. J. Kay. Wien, New York: Springer-Verlag.

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

Kozierok, R., and P. Maes. A learning interface agent for scheduling meetings. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces* (Orlando, Fla.), ed. W. Gray, W. Hefley, and D. Murray. New York: ACM Press.

Knoblock, C., 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, Wisc.). Menlo Park, Cambridge, London: AAAI Press/The MIT Press.

Konopnicki, D., and O. Shmueli. 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., D. Weld, and R. Doorenbos. 1997. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (Nagoya, Japan, August). San Francisco: Morgan Kaufmann.

Lieberman, H. 1995. Letizia: An agent that assists Web browsing. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montral, August), ed. C. Mellish. San Francisco: Morgan Kaufmann.

Lieberman, H., B. Nardi, and D. Wright. 1999. Training agents to recognize text by example. In *Proceedings of the 1999 International Conference on Autonomous Agents (Agents'99),* (Seattle, Wash.), ed. O. Etzioni and J. Müller. New York: ACM Press.

Levy, A., A. Rajaraman, and K. Ordille. 1996. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference.* San Francisco: Morgan Kaufmann.

Mellish, C. ed. *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montreal, August 1995). San Francisco: Morgan Kaufmann.

Muslea, I., S. Minton, and C. Knoblock. 1998. Learning wrappers for semistructured, Web-based information sources. In *Workshop on AI and Information Integration* (AAAI); see also www.isi.edu/ariadne/aiii8-wkshp/proceedings.html9.

Perkowitz, M., and O. Etzioni. 1995. Category translation: Learning to understand information on the internet. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montreal, August), ed. C. Mellish. San Francisco: Morgan Kaufmann.

S
R
L

SgmlQL: SGML Query Language. 1997. See www.lpl.univ-aix.fr/projects/SgmlQL/.

VanLehn, K. 1987. Learning one subprocedure per lesson. *Artificial Intelligence* 31: 1–40.

Extensible markup language (XML): Part 2. Linking. 1997. W3C Working Draft July 31, www.w3.org/TR/WD-xml-link.