From: AAAI Technical Report SS-00-01. Compilation copyright © 2000, AAAI (www.aaai.org). All rights reserved.

Programming by Demonstration for Information Agents

Mathias Bauer, Dietmar Dengler, Gabriele Paul, Markus Meyer
DFKI
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
{bauer,dengler,gpaul,meyer}@dfki.de

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 is 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.

An ideal candidate for the role of such a supporting agent is the *user* of a system 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. As a consequence it seems rational to ask her for help whenever the system gets into trouble. The paradigm of *programming by demonstration* (*pbd*) provides a feasible framework for the particular kind of dialog required in such situations in which both user and agent use their individual capabilities not only to complement each other in order to overcome the *current* problem; instead the objective is to extend the agent's skills, thus enabling him to successfully deal with a whole *class* of problems and avoiding similar difficulties—and thus, additional training effort—for the future.[1]

As a concrete application scenario imagine a Web-based travel agent that makes use of dynamic information located at various Web sites in order to configure a trip satisfying the user's preferences and constraints. Typical information 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 that are not flexible enough to deal with this unexpected situation or at least recognize the fact that there exists a problem at all.

Wouldn't it be good if this agent could tell e.g. 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 it is necessary for the agent not to remain passive and have the trainer do all the work, but instead to actively participate in the training dialog and guide the teacher to give him just the right lessons to solve his problem.

The rest of this paper is organized as follows. The next section describes a typical application scenario. Then we introduce the basic approach underlying the training dialog before we briefly describe first insights gained, discuss other potential applications, and summarize the results.

An Application Scenario

To illustrate both the kind of situation in which the above-mentioned training dialog takes place and the collaborative nature of such a session, assume our user is preparing for a trip. Using her Web browser she enters the relevant data like cities to be visited and budget limitations, leaving the rest to a Web-based travel agent that 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 has to be fetched from the Web at planning time. Examples include departure times from train or flight schedules, prices for hotel rooms and so on.

Hidden to the user, the trip planning agent formulates corresponding information requests to be answered by an *Information Broker* by querying the appropriate information sources described in a data base (compare Figure 1 for a simplified outline of the TrIAs framework for *Trainable Information Assistants*). If everything works out just fine, i.e. if all the answers to the travel agent's questions can be found on the Web, the user is presented the final result in

[1] Throughout the rest of this article we will refer to the user in the female form, while the agent will be referred to using male forms.

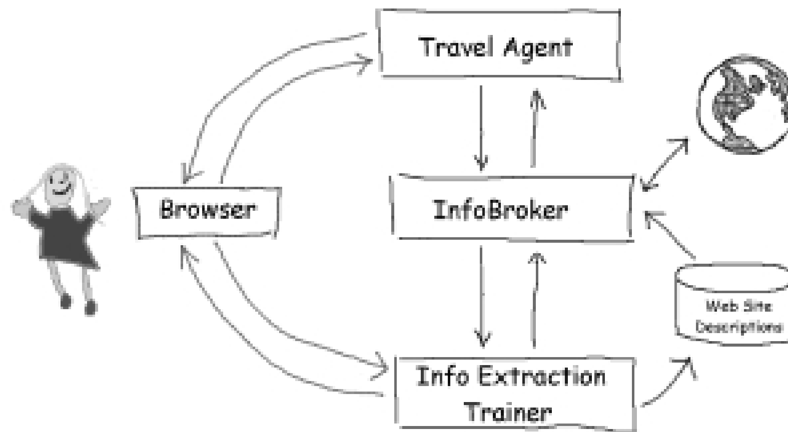

Figure 1: A typical application scenario.

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 the information extraction procedure useless that was stored in the data base containing the 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 an error message at best (unless there exist alternative sites *that have not changed recently* providing the same information).

- Some poor guy in charge of maintaining the data base of the Information Broker 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 little bit 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 just as cooperative, thus improving the overall system performance.

To be concrete, the problematic HTML document is handed over to the *Information Extraction Trainer* along with the current information request (the agent's question the answer for which 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 where some extra frames are used to guide the training dialog 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 data base for future use.[2]

The Training Dialog

The objective of the training dialog is the generation of a new information extraction procedure or *wrapper* in terms of *HyQL*, a Web query language specifically designed to support the navigation to and robust identification of relevant portions of on-line documents.

The HyQL scripts to be generated consist of a number of components most of which are optional. In general they contain

- a *selection* representing the document portion to be extracted,

- a *characterization* of the selection in terms of syntactic (HTML) features like font, style, color etc.,

- a *context* representing a larger region of the document containing the selection, and

[2]In order to avoid frequent training sessions, the Web query language HyQL(Bauer & Dengler 1999b) is used as the target language. HyQL allows both navigation through the Web and characterization of relevant document parts at a very abstract level such that many document modifications do not affect the feasibility of these procedures.

- *landmarks*, i.e. optically salient objects in the document or document parts that have a close semantic relationship to the selection.

While the first two serve the exact identification of the target concept, the latter two are intended to make the navigation through the HTML document and thus, the identification of the target concept, more robust. Typical examples for a context include tables (that can be identified by certain syntactic criteria) or "bracketed" regions e.g. between two horizontal lines (HTML tag `<hr>`). Typical landmarks are graphical objects preceding the selection or headings describing the following information (e.g. "**departure time:**").

Hope is that these navigational aids are easy to identify within the document and the path from a landmark or the beginning of a context is "simple" and unlikely to be modified (e.g. it is reasonable to assume that the close proximity between the heading "**departure time:**" and the time information following it will be preserved whatever modifications might be performed).

As this training dialog is based on the paradigm of *programming by demonstration* (*pbd*), the user's task then is to simply mark the relevant document portion with her mouse, possibly give hints as to which syntactic properties might be relevant for its reliable identification, point at appropriate contexts and landmarks, and tell the system to try and compute a wrapper using the currently available information.

In the first version of the pbd interface (described in (Bauer & Dengler 1999a)), initiative was almost completely left to the user who could ask the system to make suggestions for appropriate landmarks and contexts or validate the user's choice. Although there was no formal study on the usability of this system, it soon became obvious that the user repeatedly faced similar problems preventing her from reaching a close to optimal result (the system can *always* produce a working wrapper as long as the user correctly selected the intended document portion, but its robustness crucially depends on the careful composition of wrapper components). Broadly speaking, the user's missing insight into the system's current state made it very hard for her to choose the next step.[3]

As a consequence, the current version of the training interface (described in (Bauer, Dengler, & Paul 2000)), employs a *task library* to suggest potentially beneficial steps to the user. Each task, consisting of a number of actions like identifying a landmark, leads to the generation of a wrapper of some particular class, e.g. those dealing with information within a table or combining a syntactic characterization with a landmark.

[3] In the "Lessons Learnt" section these problems will be discussed with some more detail.

After each user action providing new information to the learning system, it checks which wrapper classes are still feasible and what type of information (and thus, user action) is required in order to advance the corresponding task. Among all candidate actions a ranking is computed according to the highest expected utility of each alternative. To compute these values, the system takes into account

- the numerical assessment v_{curr} of the "best" wrapper that could be generated using the information available so far (this computation is explained below);
- the numerical assessment $v_{new}(a)$ of the "best" wrapper that could be generated after successful execution of action a;
- the probability $p(u, a)$ of user u successfully carrying out action a;
- a penalty $annoy(u, n)$ for annoying the user with another (the $n + 1$st) action.

Remark: The abovementioned numerical assessment is intended to reflect the estimated robustness of a wrapper. In order to come up with this measure for a concrete wrapper, the system has at its disposal a library containing HyQL templates implementing wrapper components that have to be combined and instantiated in the right way. Each of these components is (heuristically) assigned a number representing the level of confidence in its robustness against document modifications. When configuring a wrapper out of these components, their initial values are combined in a way such as to adequately assess the final result. The exact computation is beyond the scope of this paper, but the basic idea is to prefer wrappers containing detailed descriptions of their target concepts over fuzzy ones and simple, short access paths over longer and more complex ones (e.g. the search for the first occurrence of some concept is preferred over the search for the 156th).

Given the above information, the expected utility of the various actions advancing one or more of the active tasks is computed as

$$util(a, u, n) = (v_{new}(a) - v_{curr}) \cdot p(u, a) - annoy(u, n)$$

If the expected increase in the wrapper quality does not justify another user action, the system suggests to finish the training dialog. In any case, all possible actions are presented to the user in an order reflecting their respective expected utility. The user, however, is totally free to ignore these suggestions. Additionally, in most cases she can choose to either carry out an action by herself or let the system do this for her and simply acknowledge the result.

Figure 2 depicts part of the interface used during the training dialog. The upper window contains the actions

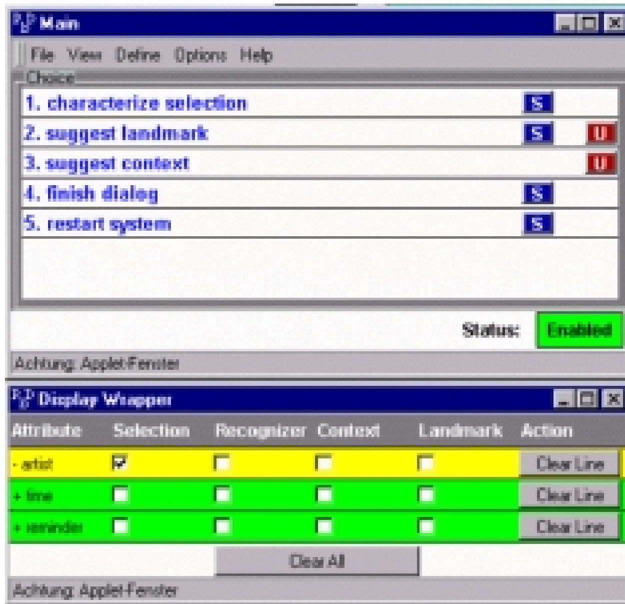

Figure 2: Part of the user interface during the training dialog.

suggested by the system where their order reflects their respective estimated utilities. The "S" and "U" buttons are used to either make the system itself perform the corresponding action or let the user take control. The lower window is used to check the constituents of the various wrappers defined for the document at hand (in the current situation the user would check which part is selected by the wrapper for concept "artist").

Besides identifying possible subsequent actions, the task models serve the purpose to estimate the effort necessary to complete a particular line of actions once the user decides to enter a subtask. That is, in each case the user is informed about the number of actions that will be necessary to arrive at a particular result (wrapper class). Finally, we observed that even with these suggestions users tend to get lost while working at some task, especially when there is no mandatory temporal order for the various steps to be carried out. So in the current version of our system we offer the user one particular linearization of the actions to be carried out whenever she enters a subtask that can only be executed as a whole.

Lessons Learnt

Although we have not yet performed a rigorous evaluation of the dialog strategies sketched above, a few general lessons could already be derived from the first prototype of the pbd environment (Bauer & 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 were observed in our own interaction with the system.

What to do next? In order 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. This is usually not the case unless she is given some tutorial about system internals that actually should be hidden from her.

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. This problem is particularly difficult in applications like the one described here where there is no uniquely characterized "goal state" that has to be reached.

To summarize, total freedom for the user is useful just in case she has sufficient background knowledge to make an informed decision. As we did not want to bother the user with too many technical details, the training dialog guidance as described in the previous section addresses the above problems by

- suggesting the potentially most effective action to be carried out next,
- proposing to finish the training session when the quality level of the wrapper to be generated is unlikely to be increased by any action,
- linearizing subdialogs even when no mandatory temporal order among its constituents exists.

Note that all system suggestions can be safely ignored by the user whenever she feels the need to do something completely different.

Other Application Scenarios

One peculiarity of the above application scenario was the exchange of the roles as service provider and consumer between user and system. This made it necessary to carefully take into account the user's "felicity" (in contrast to exclusively concentrating 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 pbd approach as described above was used to implement the InfoBeans system (Bauer, Dengler, & Paul 2000) in which even naive users can configure

their own Web-based information services satisfying their individual information needs.

As this application is in the tradition of pbd-systems like (Lieberman, Nardi, & Wright 1999) where 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 some system suggestion. Experiments will show which of these versions will have the better user acceptance.

Conclusions

This paper described how task models and a personalized decision-theoretic quality measure can be used to guide a user through an interaction in which no explicit goal states exist. The programming-by-demonstration framework presented is adaptable to various application scenarios, thus enabling the system designer to take into account various possible roles of the user.

First informal tests with non-expert users indicate that the training mechanism provided enables (many) end users to successfully deal with delicate problems of identifying and extracting information from Web-based information sources. Besides the two application scenarios sketched above—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

Bauer, M., and Dengler, D. 1999a. InfoBeans – Configuration of Personalized Information Services. In Maybury, M., ed., *Proceedings of the International Conference on Intelligent User Interfaces (IUI '99)*, 153–156.

Bauer, M., and Dengler, D. 1999b. TrIAs: Trainable Information Assistants for Cooperative Problem Solving. In Etzioni, O., and Müller, J., eds., *Proceedings of the 1999 International Conference on Autonomous Agents (Agents'99)*, 260–267.

Bauer, M.; Dengler, D.; and Paul, G. 2000. Instructible Agents for Web Mining. In Lieberman, H., ed., *Proceedings of the International Conference on Intelligent User Interfaces (IUI 2k)*. to appear.

Lieberman, H.; Nardi, B.; and Wright, D. 1999. Training Agents to Recognize Text by Example. In Etzioni, O., and Müller, J., eds., *Proceedings of the 1999 International Conference on Autonomous Agents (Agents'99)*, 116–122.

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