Trainability: Developing a responsive learning system

Steven N. Minton

Fetch Technologies, Inc.
4676 Admiralty Way
Marina del Rey, CA 90292 USA
+1 310 448 8275
Steve.Minton@fetch.com

Sorinel I. Ticrea

Fetch Technologies, Inc.
4676 Admiralty Way
Marina del Rey, CA 90292 USA
+1 310 448 8425
Sorin.Ticrea@fetch.com

Jennifer Beach

Fetch Technologies, Inc.
4676 Admiralty Way
Marina del Rey, CA 90292 USA
+1 310 448 8223
Jennifer.Beach@fetch.com

Abstract

In this paper, we describe the lessons we learned in developing AgentBuilder, a commercial system for rapidly creating agents that extract information from web sites. AgentBuilder employs a Programming-by-Example (PBE) paradigm, where users train the system by showing the system examples of web pages and the information to be extracted from these pages. The system uses a sophisticated machine learning algorithm for inducing extraction rules from examples and eventually creates web agents that navigate through a site and extract information. Previous work on Programming-by-Example has discussed the importance of *felicity conditions*, which simplify training so that the learner can more readily understand what is being taught. In this paper, we show that, in addition, developers must design the learning system to insure that the teacher (i.e. the user) can train the system without becoming frustrated. We discuss these characteristics, which we refer to as "trainability", and show how trainability considerations constrain the type of learning algorithm that can be employed in a practical AI application.

Introduction

Due to advances in networking technology, one can now obtain access to large numbers of remote information resources via private intranets or the public Internet. However, the Web's browsing paradigm does not support many information management tasks that involve monitoring, filtering, aggregating and/or integrating information. In response to this problem, we have developed a commercial system -- the Fetch Agent Platform -- for creating and executing Web agents that can carry out a wide variety of such tasks. The platform is descended from research on information agents (Knoblock, Lerman, et al. 2000) originally carried out at USC's Information Sciences Institute. It consists of several components, including AgentBuilder, a design time component for specifying Web agents, and AgentRunner, a high-performance dataflow system for executing agents.

In this paper we describe the lessons we learned while developing and deploying the user interface for AgentBuilder. An important aspect of AgentBuilder is that it *learns from examples*. Specifically, the user instructs the system how to navigate through a Web site and extract information. For example, suppose that a user wants to create an agent that can extract book titles, authors, and prices from a book sellers site like Amazon.com. Using a GUI built on top of Internet Explorer, the user demonstrates how to extract examples of titles, authors, and prices from sample pages on the site.

This example-based approach is particularly appropriate for building Web agents. Given that the user knows exactly the data that s/he wants to retrieve, it is much easier for users to pick out examples of the data than write rules for navigating and extracting the data. Because of this natural fit, over the past few years many researchers have developed machine learning algorithms for wrapper induction (i.e., learning grammars for web pages based on labeled examples) and similar information extraction applications (e.g., Kushmerick et al, 1997; Bauer et al., 2000; Muslea, Minton, and Knoblock 2001; Laender et al, 2002). The focus in this paper is not the details of the induction algorithm used in AgentBuilder (see Muslea, Minton, and Knoblock 2000, 2001), but on the interaction with the user as s/he trains the system.

In previous work on Programming-by-Example (PBE) (and Programming-by-Demonstration) it has been noted that PBE systems can take advantage of *felicity conditions* (VanLehn, 1987; Maulsby et al., 1989), i.e., common conventions in human-human interaction that teachers use to reduce the difficulty that their pupils face during learning. For example, when instructing a student how to solve a problem, a teacher should *introduce at most one new branch per lesson* (Van Lehn, 1987). Like previous PBE systems, AgentBuilder is designed to lead the teacher (i.e., the user) to obey such conventions, in order to simplify the learning task for the induction algorithm.

In this paper, we point out that there are also a set of "inverse felicity conditions" that a learner must obey in order to help out the teacher. For instance, the learner should take a "relatively short time" to respond to the teacher, even if it means giving up on a potentially productive search. We refer to these conditions in aggregate as *trainability*. Many of these aspects of trainability are commonsensical, but as we will explain,

these considerations lead to some interesting constraints on the types of learning algorithms that are useful in practice.

AgentBuilder Overview: The User Experience

In this section we will outline the basic capabilities of AgentBuilder and show how the user can rapidly create a Web agent that extracts information from a site. We will use the book-purchasing site Bookpool.com for our running example. Assume that the user wants to create an agent that takes a keyword as input, and returns detailed information about each book that matches the keyword. The entry page for the Bookpool site has a keyword-based search form. If a user submits a keyword such as "Java" the site will return a list of books that match this keyword. For each book there is a "details page" that shows the book's cover and lists the title, price and other relevant information. Finally, from each details page there is a link to an author page, which provides information about the author of the book (e.g., other books by that author).

The Agent Layer

The user begins the agent building process by specifying how the site is designed in terms of the different page types. For each page type, the user creates a "wrapper" that specifies how to process the data that appears on that page type, and how to navigate to the next page type.

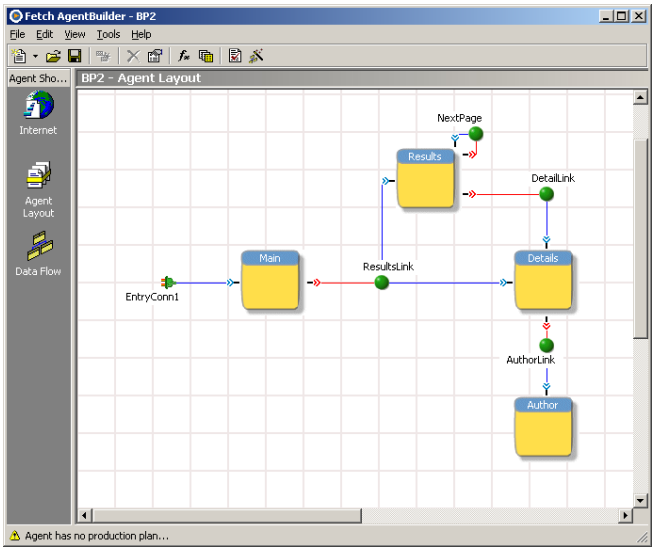

Figure 2. The agent for Bookpool site.

The figure above illustrates how AgentBuilder displays an agent to the user. In this agent overview (also called the "Agent Layer" view), each box corresponds to a wrapper for a particular page type, and each connection between the wrappers corresponds to a navigation action (a link or form) within the site. The agent consists of four wrappers, one for the search page (the **Main wrapper**), one for list pages (the **List wrapper**), one for details pages (the **Details wrapper**), and one for book's author information (the **Author wrapper**).

There is also a "more link" which provides the next twenty-five matches in response to the given query (note that the "more link" connects from a list page to another page of the same type). Finally, there is a new connector between the search page and the details page, indicating that some keyword searches bypass the list page and return a details page (e.g., such as when the keyword is an ISBN).

Users can easily create and name new wrappers and connect them together. To specify the details of each wrapper, the user must switch modes to the wrapper view, "entering" a wrapper by clicking on the appropriate box.

The Wrapper Layer

To specify the behavior of an individual wrapper, the user leaves the Agent Layer by double clicking on a wrapper and entering the wrapper view (see Figures 3 and 4). The process of specifying a wrapper consists of five stages. First, the user downloads a set of sample pages. This is a very simple task, since wrappers are built in sequence, and the connector from the previous wrapper can be used to automatically download the sample pages. For example, to download sample pages for the **Details wrapper**, AgentBuilder will use the URLs extracted by the previous wrapper, that is, the **List wrapper.** AgentBuilder makes this workflow simple by providing a simple wizard-like interface to accomplish this task.

Next, the user declares a description of the data to be extracted from the page. As shown in Figure 3, AgentBuilder enables users to construct a Data Declaration Tree through a wizard-like interface. The Data Declaration Tree is essentially a simplified XML schema describing the structure and attributes of the data targeted for extraction. For example, the Details wrapper in Figure 3 extracts specific information about a book, such as its title, ISBN, and price. When this wrapper is executed, it will return an XML document with the same structure specified by the tree shown on the left-hand side of the screen.

The user *trains* the learning system by marking up sample data for the learning system, in effect, instantiating the Data Declaration tree on selected sample pages. To do this, the user selects examples of the fields on a sample page, and drags-and-drops the data on the tree, as shown in Figure 4. The user then invokes the learning system in order to produce a set of extraction rules that will automatically extract the targeted data from all of the pages belonging to the wrapper's page type.

In the next phase, the learned extraction rules can be *tested* by automatically applying the rules to the other sample pages. If the rules are not extracting data as expected, the user can correct the rules by providing additional samples to the learning system, and re-generating the rules. The user continues this iterative process until the extraction rules are correct. AgentBuilder makes it easy to test the learned rules rapidly over a large number of sample pages. The result is that highly reliable rules can be efficiently produced.

In the final phase, the *connection* phase, the user specifies how to connect the current wrapper to subsequent wrappers by generating HTTP requests that retrieve the next set of targeted pages. Requests can be generated either from forms, or from links. For example, for the Bookpool.com **Entry wrapper**, we use the search form on the left corner of the page to generate requests for list or detail pages. For the **List wrapper**, we use the URLs extracted from the page to generate requests for details pages. Again, AgentBuilder provides easy wizard-like interfaces that enable users to specify how to use the forms or links to generate the necessary HTTP requests, without knowing any technical knowledge about HTTP *per se*.

Trainability

Teaching a lesson involves a series of interactions between a student and teacher. Many researchers have worked on computer tutoring systems, where the main question is how a computer tutor should interact with a human student to facilitate learning. In Programming-by-Example (PBE) systems (and similarly, Learning Apprentice systems) we study the opposite situation, where the computer is the student and human is the teacher. Much of the work on PBE systems has focused on the design of graphical user interfaces that make it easy for a human teacher to convey complex examples to the system (Cypher, 1993).

In designing AgentBuilder, we certainly devoted a great deal of attention to the GUI to ensure that it would be easy for the user to program by example. As noted in the previous section, an agent is broken down into a series of simple wrappers, each of which is separately programmed. Examples of what each wrapper should extract are easily created using a drag-and-drop paradigm.

However, AgentBuilder is distinguished from most previous PBE systems because the learning algorithm is by necessity sophisticated and non-intuitive. In most previous PBE systems, the user demonstrates a series of operations, and the system essentially learns a macro operator. As discussed by Witten (1995) and Lau and Weld (1999), the generalization process carried out by many PBE systems is simplistic (In fact, the point is to make the generalization process appear trivial to the user so that s/he can easily understand what the computer has learned (Minton et al., 1995)). This is not possible in AgentBuilder, because the human teacher does not know how to solve the problem. The extraction rules learned by AgentBuilder are not necessarily obvious; in practice the system typically traverses a large search space to derive reliable extraction rules that work across all the pages. The human user only knows that the pages have a regular structure – s/he will not normally be able to describe the structure precisely.

Thus, a critical distinction here is that the human teacher does not necessarily understand the extraction rules produced by the system, and moreover, the teacher almost certainly will not appreciate the difficulty of the learning process. This complicates the interaction between teacher and student. In the end, we found that in order for the interaction between the teacher and student to be successful (from the teacher's point of view), certain conditions must be satisfied, which we refer to as trainability. These conditions are analogous to Van Lehn's *felicity conditions* that are necessary for a teacher to instruct a system successfully, that is, to show all steps, introduce at most one new branch per lesson, etc. (Van Lehn, 1987).

We have identified the following three conditions as critical to trainability:

1) Clear progress: The student should learn rapidly, requiring "relatively few" examples. The exact number of examples required can depend on the teacher's perception of the difficulty of the task. If a task is perceived to be simple, the teacher will lose patience rapidly. While this trainability condition seems obvious, it leads to some stringent requirements on the learning algorithm in practice. For example, it suggests the importance of active learning techniques, where the computer selects the examples that would be most profitable for learning. We found in AgentBuilder that it was imperative for the interface to enable the user to identify the pages s/he should choose to mark up as training samples. Without this ability, the user will spend his or her time unproductively, marking up pages that fail to help the learning system improve, and becoming rapidly frustrated. (We will discuss this more concretely in the next section.) We also found that the learning system should immediately make generalizations that are obvious to the user. This can be difficult, because what is obvious to the user may not be obvious to the learning system. The only solution to this is to give the system a strong heuristic bias (ideally, keeping the sample complexity low). Finally, *clear progress* also implies that the system should **never** make mistakes on examples that the user has already marked up. Not all learning algorithms provide this guaranty. In fact, this was a problem in the early versions of AgentBuilder, because our learning algorithm initially used heuristics that maximized expected performance, even at the expense of failing on some of the pages that the user had marked up. While this seems like a minor problem to the machine learning researcher (since with more examples the system will ultimately find a perfect rule if one exists), we found that users became extremely annoyed when the system would make mistakes on examples they had provided. It would also cause users to question the learning system's competence.

2) Concrete feedback. During the training process, the student should provide immediate feedback to the teacher so that the teacher can rapidly determine where the student is making mistakes and rectify the problem. As we noted previously, AgentBuilder uses a sophisticated learning system, and most users will not understand how it searches its hypothesis space. Indeed, we found that most users do not even bother to look at the extraction rules that are learned, because they are difficult to understand. (The rules are similar to regular expressions.)

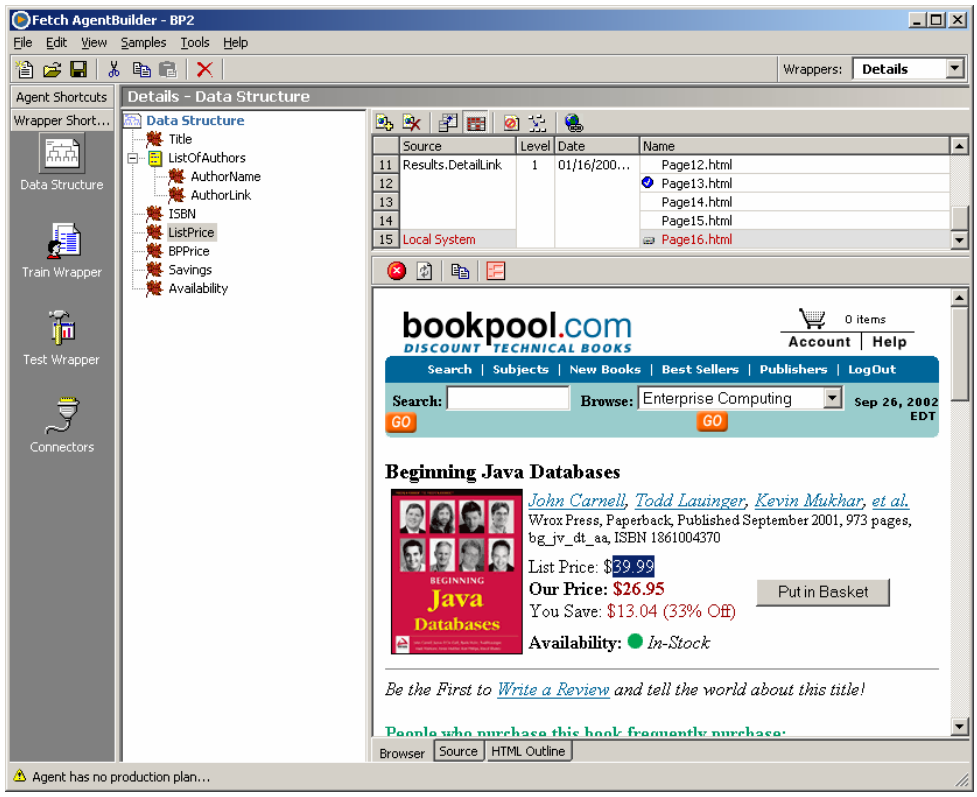

Figure 3. Data Declaration Tree for the Details wrapper.

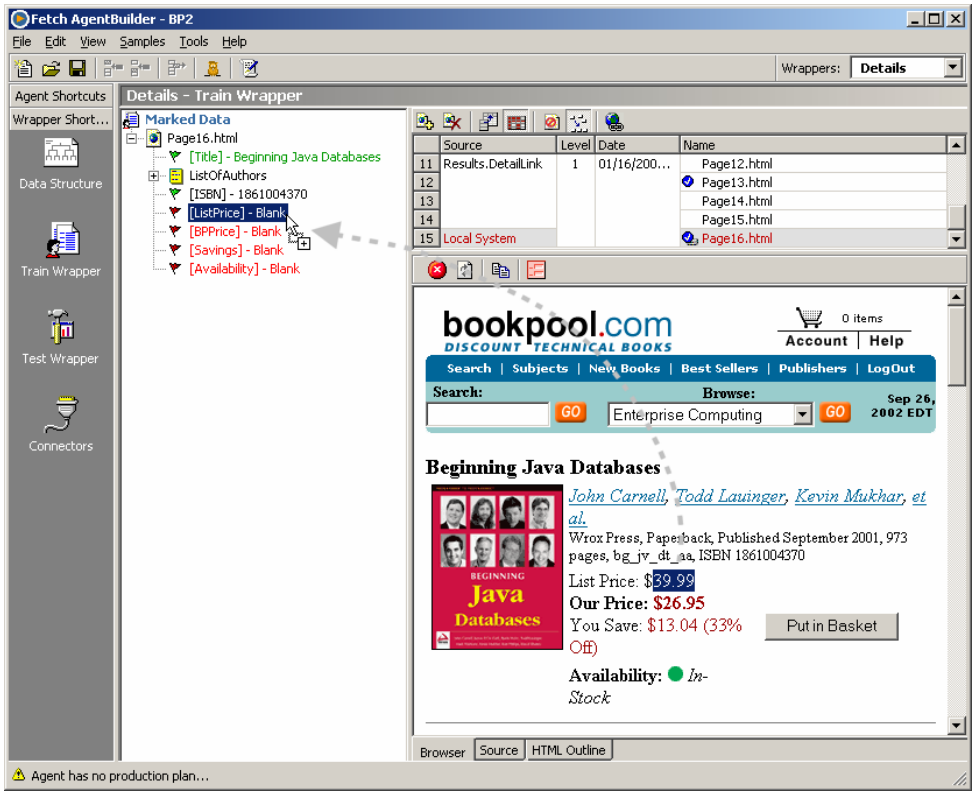

Figure 4. The user indicates data of interest by using the drag-and-drop technique.

As a result, the question of how to provide clear feedback is nontrivial. We experimented with various feedback mechanisms in AgentBuilder. In the end, we settled on an approach that makes it easy to *test* learned rules on large numbers of sample pages in order to gauge how well rules are working. If an extraction is failing on a given page, the user can then mark up that page to provide additional examples and rerun the learning mechanism.

The following capabilities are vital for the testing process. First, the system makes it extremely easy to download large numbers of sample pages. Second, users can define *validation rules* that are used during the testing process to flag examples where extracted content is suspicious. Validation rules describe the expected characteristics of the extracted data. For example, a validation rule might check to make sure the rule is extracting a number, or if the extracted data contains any HTML tags. Extracted content that fails validation is highlighted in the GUI. Third, the learning algorithm and the testing process are both designed to operate very quickly, so that the user can rapidly cycle through training and testing. After marking up some examples, the user can call the learning algorithm, test the results, select one or more examples to correct, and rerun the learning algorithm.

3) Fault tolerance: No one is perfect. The learning system must be designed to be sensitive to the fact that users are human, and in particular, that they have limited patience and may make mistakes during training. As noted previously, the learning system heuristically explores a potentially large search space to find a good extraction rule. If the learning algorithm is having trouble finding a rule, it will, if left to its own devices, continue searching for an arbitrarily long time. We found that users tended to lose patience rapidly, and that overall we were better off setting a relatively short time limit for the learning system. As a consequence, the learning process will occasionally time out unsuccessfully in cases where it would have found a rule if it had run for a longer period of time. Although, from an academic point of view, accuracy may be paramount, from our users' point of view a slight decrease in accuracy is preferable compared to the annoyance of having the system occasionally search for unbounded periods of time.

Users are not only impatient, but they also often make mistakes during training, and the system must also be designed to be tolerant of their mistakes (Kaiser et al., 1995). Typically these are markup errors. For example, the user may grab slightly too much text during the drag-and-drop process (such as including spurious punctuation following the target string), or, when marking up the items in a list the user may grab the fourth item instead of the third item by mistake. When faced with this "noisy training data", our learning algorithm will do its best to find a rule which extracts exactly what the user specified. This can lead to unexpected and inexplicable results from the users point of view (since the user will not normally realize that he has made a mistake). When this happens, novice users are quick to blame the learning algorithm, rather than checking their own markup. To improve the user experience, we added several methods for detecting anomalies. For example, if a learned rule is disjunctive, and one of the disjuncts covers only a single example (typically indicative of a markup error) AgentBuilder will warn the user. Similarly, if the system detects that the examples in a list do not occur on the page in the same order that they do in the list, it will warn the user.

The Long and Winding Road

The principles described in the previous section seem obvious, but developing AgentBuilder was an arduous process and some of the mistakes we made along the way are instructive to recount.

The question of how the learning system should give feedback to the user is one we have worked on for many years. Our research on wrapper induction algorithms prompted us to develop an active learning technique called Co-Testing (Muslea, Minton, et al. 2000). In active learning, the computer selects training examples for the user to label. By selecting the most informative examples for the user to label, active learning can significantly reduce the number of training examples required. The idea behind Co-Testing is to learn two (or more) independent rules. Each rule is separately learned in an independent hypothesis space, so Co-Testing is a type of *multi-view learning*. Then we test the rules on the unlabeled examples. If the rules disagree on an example, it becomes an informative example for the user to label. In the context of AgentBuilder, this technique can be applied as follows. The system learns two different extraction rules, one that describes how to find the desired content working forward from the top of the page, and another that describes how to find the desired content working backward from the bottom of the page. If the rules extract different content from the same page, then the user should be asked to mark up the content on that page. Based on the proportion of pages on which the two rules agree, we can also generate a confidence metric, which tells the user how well the learning system is doing.

We prototyped this scheme in AgentBuilder, but users were stymied by the complexity. Rather than showing them two different learned rules, we elected to show them just the forward rule to shield them from needing to understand co-testing. However, this strategy backfired because if the forward rule was completely correct, while the backward rule still had some bugs, the user would be asked to label pages even though there was no apparent need for more training (since the forward rule was extracting correctly).

In the end, our experience with co-testing led us to the simpler scheme outlined in the previous section, where validation rules are used to detect extraction problems that are then highlighted for the user. This scheme does not impose a high burden on the user because the system can

automatically produce a default set of validation rules without any user actions.[1] Note that this scheme is really a simplified version of co-testing, since an automatically learned validation rule constitutes a second independent rule for testing.

Another issue with the early versions of the system is that we did not pay enough attention to the fact that mistakes made by the teacher were a major source of problems. Instead, we focused on improving the accuracy of the system, assuming that the teacher behaved perfectly. However, users make all sorts of mistakes, such as incorrectly marking up pages, or failing to understand the basic concepts of the GUI. We have since significantly improved the system by streamlining the responsibilities of the user. Currently, the user is led through the sequence of actions s/he must perform Wizards are used whenever possible, in order to reduce the user's cognitive load.

We also improved the training process to reduce the likelihood of user mistakes during the drag-and-drop process. Improvements include issuing a warning to the user by flagging markup anomalies, and providing the user with a variety of different types of feedback during the drag-and-drop process. For example, the user can click on any training example that s/he has previously marked up, and the system will highlight the text on the page (in both the browser and the source HTML).

Conclusion

Our work has demonstrated how machine learning can be incorporated within a user interface to automate the process of creating extraction rules. However, machine learning researchers have primarily focused on improving the accuracy and efficiency of learning. As we have argued, careful attention must also be devoted to trainability, i.e., to designing an interface with which users can interact in a rewarding manner.

Acknowledgments

We thank Cenk Gazen and Ion Muslea for their work on the core learning algorithm used here. In addition, numerous other individuals contributed materially to the design and implementation of AgentBuilder, particularly Bryan Pelz, Gary MacKinnon, and Craig Knoblock. The work here was funded in part by the NSF under contract DMI-0090978, the Air Force under contract F30602-02-C-0029, and DARPA under contract F30602-01-C-0197. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

[1] Examples of default validation rules include "Do not allow null data", "Do not allow HTML tags in the extracted data", and "Must be a valid URL".

References

1. Barish, G., DiPasquo, D., Knoblock, C. A.., and Minton S. N. 2000. A Dataflow Approach to Agent-Based Information Management. In *ICAI Proceedings*, Las Vegas, Nevada, USA.
2. Bauer, M., Dengler, D., and Paul, G. 2000. Instructible information agents for web mining. In *Proceedings of the 2000 Conf. on Intelligent User Interfaces*.
3. Cypher, Allen, eds. 1993. *Watch What I Do: Programming by Demonstration*, MIT Press.
4. Kaiser, M., Friedrich, H., and Dillmann, R. 1995. Obtaining good performance from a bad teacher. In Workshop on Programming by Demonstration, International Conference on Machine Learning.
5. Knoblock, C. A.., Lerman, K., Minton, S. N., and Muslea, I. 2000. *Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach*. IEEE Data Engineering Bulletin 23 (4), 33-41
6. Knoblock, C. A.., Minton, S. N., Ambite, J. L., Ashish, N., Muslea, I., Philpot, A. G., and Tejada, S. 2001. The Ariadne approach to web-based information integration. *Special Issue on Intelligent Information Agents: Theory and Applications*, 10(1/2), *IJCIS* 2001, 145-169.
7. Kushmerick, N., Weld, D., and Doorenbos, R. 1997. Wrapper Induction for Information Extraction. In *IJCAI Proceedings*, 729-735. Nagoya, Japan.
8. Laender, A., Ribeiro-Neto, B., Silva, A. and Teixeira, J. 2002. A Brief Survey of Web Data Extraction Tools, SIGMOD Record, Volume 31, Number 2.
9. Lau, T. and Weld, D.S. 1999. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the International Conference on Intelligent User Interfaces*, 145–152.
10. Maulsby, D.L., Witten, I.H., and Kittlitz, K.A. 1989. Metamouse: Specifying Graphical Procedures by Example. Computer Graphics 23 (3).
11. Minton, S. N., Philpot, A. G., and Wolfe, S. 1995. Specification-By-Demonstration: The ViCCS Interface". In Workshop on Learning from Examples vs. Programming by Demonstration.
12. Muslea, I., Minton, S.N., and Knoblock, C. A. 2001. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4: 93-114.
13. Muslea, I., Minton, S. N., and Knoblock, C. A. 2000. Selective sampling with redundant views. In Proceedings of AAAI-2000.
14. Van Lehn, K. 1987. Learning one Subprocedure per Lesson. *Artificial Intelligence*, 31:1–40.
15. Witten, I. H. 1995. PBD systems: when will they ever learn?. In *Workshop on Programming by Demonstration, ML'95*, 1-9. Tahoe City, Calif.