A Research Agent Architecture for Real Time Data Collection and Analysis

Travis Bauer
Computer Science Department
Lindley Hall, Indiana University
150 S. Woodlawn Avenue
Bloomington, IN 47405, U.S.A.
trbauer@indiana.edu

David Leake
Computer Science Department
Lindley Hall, Indiana University
150 S. Woodlawn Avenue
Bloomington, IN 47405, U.S.A.
leake@indiana.edu

ABSTRACT

Collecting and analyzing real-time data from multiple sources requires processes to continuously monitor and respond to a wide variety of events. Such processes are well suited to execution by intelligent agents. Architectures for such agents need to be general enough to support experimentation with various analysis techniques but must also implement enough functionality to provide a solid back end for data collection, storage, and reuse. In this paper, we present the architecture of Calvin, an research grade agent system for supporting and analyzing users' document access. Calvin provides specific applications for collecting, storing, and retrieving data to be used for information retrieval, but its extensible object oriented implementation of resource types makes the architecture sufficiently flexible to be useful in multiple task domains. In addition, the architecture provides the ability to capture and "replay" data streams during processing, enabling the automatic creation of data testbeds for use in experiments comparing alternative analysis algorithms.

1. INTRODUCTION

Calvin is a Java-based agent system for researching agent-based personal information retrieval. Calvin monitors users while they are using a computer, recording and analyzing the documents they access. It then indexes resources accessed and suggests previously indexed documents, based on their relevance to the user's current task. Developing infrastructures for intelligent agents that monitor and support users presents two broad problems. The first is how to perform data accumulation and storage. An infrastructure needs to be in place to collect data by recording the user's actions and organize that information for convenient analysis. The second broad problem is how to use the data to identify and satisfy user needs. In Calvin, this corresponds to how to analyze the documents accessed in order to identify task contexts, and how to index new documents by context to assure proper future retrieval.

Calvin's Java-based infrastructure focuses on solving the former

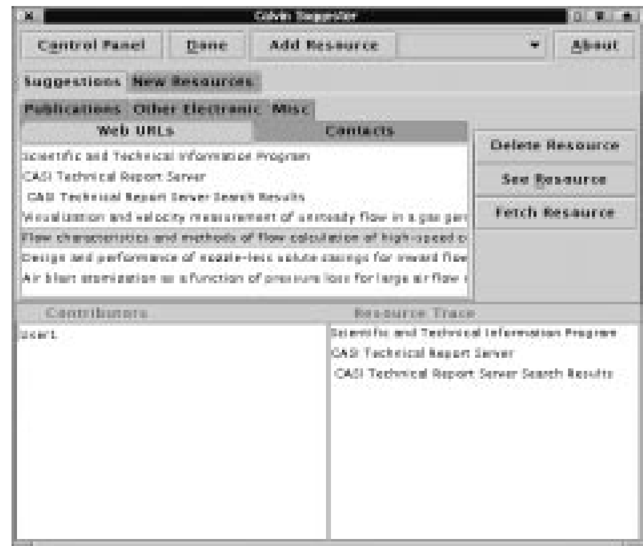

Figure 1: Main User Interface

problem in such a way that researchers can focus on solving the latter. This lets Calvin serve as a research grade agent system which can serve different data collection and analysis tasks beyond those being investigated in our research. Calvin's architecture provides a general purpose framework for collecting and recording data of customizable types. By providing a flexible interface at various levels, it is easy to change and upgrade individual parts of Calvin without substantial modifications to the entire architecture.

Central to Calvin is the ability to capture and replay streams of collected data. This enables researchers to save and reuse standardized test beds of data streams. This allows fair comparison among different analysis techniques, because they can be applied to exactly the same set of data.

2. OVERVIEW

Calvin has been designed and implemented in the context of supporting users as they access documents. As the user browses, Calvin records the documents which the user is accessing. From these documents, Calvin uses information retrieval techniques to create an index for each document reflecting its subject matter. These documents are retrieved and suggested to the user when similar documents are accessed in the future. A screen shot of Calvin's resource

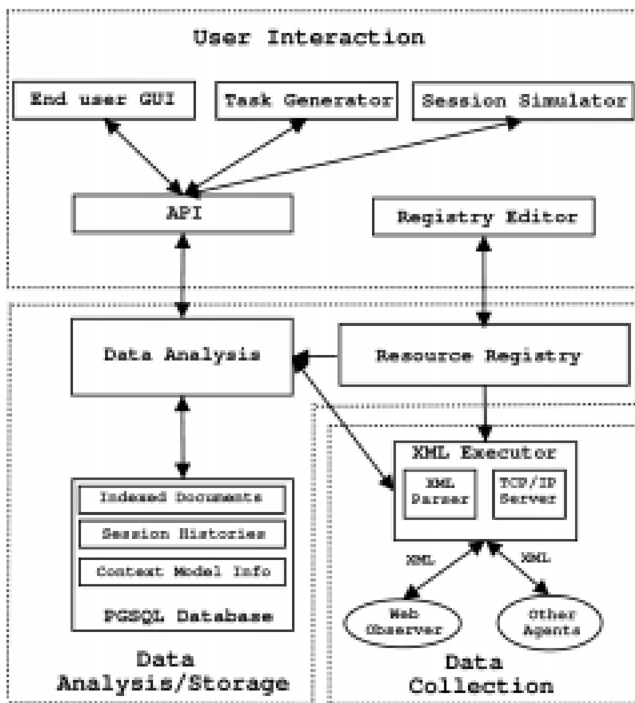

Figure 2: Calvin's Architecture

suggester is shown in figure 1. Calvin's task and user interactions are described in detail in [8].

Although Calvin was designed in the context of information retrieval, its architecture was designed to be general enough to enable using its components in multiple task domains, and for its data storage techniques to be applied in other research contexts where processing data needs to be collected and analyzed for system evaluation.

Calvin's architecture is shown in figure 2. Calvin includes subagents as part of an extensible system to collect data from various sources and send the information for analysis via an XML message. This information is analyzed by a researcher-defined "Data Analysis" component. This component conforms to a specific interface specification so that data analysis components can be changed out without changing the rest of the system. The kinds of data which can be passed among system components are specified in a registry and are customizable. This configuration allows researchers to use Calvin to perform the data collection/storage and focus on issues of analysis.

Calvin's architecture has the following features:

1. Data Collection
   (a) Flexible data collection techniques for knowledge engineering or machine learning focused approaches to information analysis.
   (b) Information collection from a diverse extensible set of resource types.
   (c) The ability to inter-operate in a multi-agent/multi-platform data collection environment using XML.
2. User Interaction
   (a) An agent user interface for making suggestions to users and allowing the user interaction.
   (b) A user interface for gathering data for controlled experiments.
3. Data Analysis
   (a) Creation/use of a test bed of document access behavior for standards creation.
   (b) Facilities for simulating/replaying user browsing.
   (c) Plug-and-play data analysis component interfaces for experimenting with multiple types of analysis without altering the infrastructure.
4. Implementation
   (a) Written in Java for cross-platform development/experimentation and uses a number of freely available packages.
   (b) Uses standard protocols for exchanging information for easy data analysis in third party packages.

The following sections discuss each of these in turn.

3. DATA COLLECTION

Calvin supports a distributed multi-agent approach to data filtering. The "XML Executor" module accepts connections via TCP/IP for the transmission of information about new documents. Agents which monitor the user communicate their information via XML. To report a document access, the monitoring agent opens a connection to the XML Executor and sends an XML message containing the document information. The Executor then replies with a message indicating whether or not it was able to parse the message.

Currently, our primary monitoring agent is a Java proxy server which monitors WWW browsing behavior and reports the pages accessed. However, using XML over TCP/IP allows non-java agents to easily interface with Calvin, and allows a potentially large number of different agents to gather information without requiring any further modifications to Calvin itself. For example, Windows API calls provide much information about user document access. Planned future development of Calvin includes writing an agent to access these API calls and send document information to Calvin.

Two way communication is supported in Calvin to eventually enable agents to send queries through the XML Executor to learn about the user, and to allow user interfaces to control the subagents. Currently our XML specification does not provide this syntax, but it could easily be added.

Although the primary kind of user resource access we are currently studying is WWW browsing behavior, Calvin is extensible to handle different resource types, including document references manually entered by users. This extensibility is handled via the Resource Registry. Using a predefined interface, programmers can write Java resource handlers and add references to them in the registry. A resource handler defines how to communicate information about a document internally among Calvin resources, user interfaces for displaying/editing meta information about the resource, and how to automatically retrieve the resource referred to.

These Java classes are registered with the Resource Registry through the Registry editor. Once they are registered, Calvin uses the registered information to determine how to treat references to the specified type of document. Because the resource handlers implement predefined interfaces, no modifications are necessary to the Calvin source code to deal with these resources; the new resource types are seamlessly integrated with Calvin. This makes Calvin customizable to different kinds of environments where different document access recording goals may be required. To support our research, Calvin has pre-defined default handlers for the following types of documents: WWW Document, Email address, Phone number, Book, and electronic concept maps [10] (a type of graphical knowledge representation).

The following illustrates the use of the resource registry. When Calvin is running and the WWW proxy server sends it an XML message, the proxy server identifies its resource type by including a field "<type>0</type>." Before further processing of the message, Calvin asks the resource registry for an instance of a class corresponding to resource type id 0. The registry responds with an instance of the *WebURLResource* class, which is a descendent of the *ResourceModel* class. Calvin then parses the rest of the XML message using the API specified by *ResourceModel*. Once the message is fully parsed, the instance of *WebURLResource* can be accessed to manipulate the resource. Section 7 provides more information on the primitives for manipulating resources.

4. USER INTERACTION

User interaction with Calvin occurs through one of three different interfaces. The "User Interface" is designed to be used by an end user in day to day interaction with Calvin. The user interface used in our research is shown in figure 1. Through this interface, Calvin suggests resources to the user. The user can also see a list of the resources Calvin has accumulated from the current browsing session. Users can set up customizable filters to prevent indexing particular kinds of documents (such as documents containing the user's name, or advertisements from web browsers). Different data analysis components may also have various user-controlled parameters, or special diagnostic displays to show the user. The user interface is designed such that different data analysis component can provide different displays to the user without requiring changes to the user interface itself.

When performing experiments to study the behavior of a data analysis technique, one may not want the user to receive so much information, or have such control over the system. In addition, it may be necessary to request additional user feedback. To facilitate collecting data for experiments, Calvin includes a "Task Generator" which provides a different user interface to the Calvin system. This interface presents the user with a series of tasks which the experimenter determines beforehand. After the user spends some specified amount of time on the task, a new window is displayed to ask the user experimenter-defined questions. Figure 3 shows a sample survey developed for an experiment on user browsing behavior.

During the experiment, the system keeps records not only of responses, but also of the task context in which those responses occurred. Information about every document accessed is gathered and stored, including the document location, full text of the document (if available) and document title, and a time stamp when the document was accessed. This information, along with the answers to the surveys are cross indexed according to an internally assigned id number and unique document id.

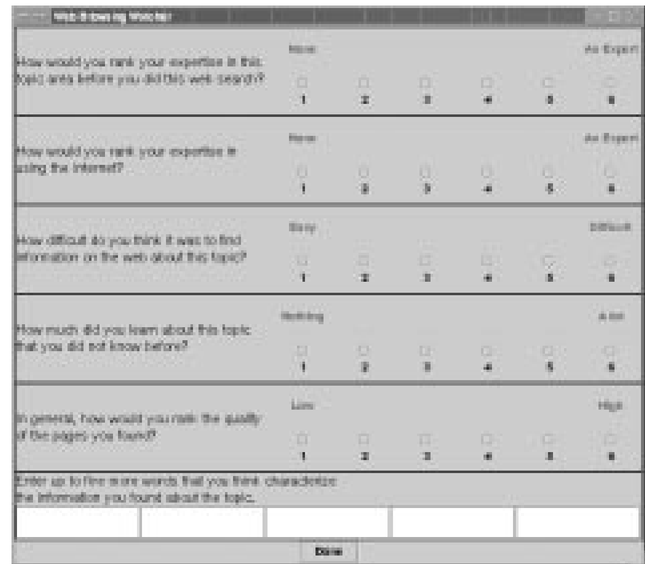

Figure 3: Task Generator Interface

5. DATA ANALYSIS

At the core of an information retrieval agent is the ability to analyze, index, and retrieve documents in a useful way. Calvin's architecture provides the infrastructure for manipulating documents. This frees the researcher from thinking about how to integrate the resources with Calvin to focus on the more interesting research question of how the data in the given task domain should be analyzed. Calvin provides "Plug-and-Play" data analysis modularity so researcher can experiment with different data analysis techniques with minimum effort.

An abstract Java class has been defined which performs most of the background work in storing and retrieving documents. This class also tells the rest of Calvin how to interact with any future data analysis component. When this abstract class is extended, different data analysis techniques can be implemented to control the behavior of the data analysis component. Because the extended classes obey the predefined interface, Calvin can use any such data analysis component without modification. Taking advantage of Java's dynamic class loading, the end user GUI and the Task Generator take the data analysis component as a parameter on the command line when the program is executed. This flexibility lets a developer experiment with different kinds of data analysis component without having to continually change the rest of the code. This frees the experimenter to deal with the research issues of information retrieval and not the technical details of dealing with a database back end for the user interface.

6. REPLAYING DATA STREAMS

Both the end user GUI and the Task Generator can automatically collect data streams, which in the context of information retrieval are sequences of document accesses. The system stores the time stamp, and full document text, providing the information needed for programs to "replay" the web search to the data analysis component. In Java, this replay is implemented as an Iterator. After specifying the path the user wants to replay, going through a past instance of user behavior is as simple as creating a *for loop*.

The ability to replay user behavior provides an important advantage

for research in personal information agents: it allows the creation of a test bed of data accumulated from user interactions with the system, to use as a standard for comparing different kinds of information retrieval. Information retrieval agents are sometimes tested on manually-collected test sets of documents, which are used to simulate document access. The Reuters collection, a set of pre-categorized new articles, is an example of such a test set. One problem with such test sets is that the documents are already sifted and organized and thus do not represent the same kind of information stream that occurs in real user behavior. Passing a set of documents from the Reuters collection through an agent is much different from providing the actual text that gets passed to a web browser during information search. Consequently, results from testing an agent against the Reuters collection may not be indicative of expected performance during actual use. However, always relying on test with real users is problematic as well: it is unrealistic to test real users on hours of web searching to test small improvements to the algorithm. Even if subjects were available, variation in the documents accessed would reduce certainty that any change in performance was due to changes in the algorithm rather than variations is user behavior.

The ability to capture test beds of past document access solves this problem. The stored documents are what users really accessed (even the advertisements can be stored), and the same test bed can be sent to multiple versions of IR algorithms. This assures that when new versions of the algorithm are tested, any changes in the performance of the algorithm are due to the algorithm, and not just variation in user behavior.

For example, we are currently developing textual analysis techniques to automatically identify keywords which serve as topic identifiers for specific user interests. The textual analysis code has undergone multiple revisions as we explore various approaches. By wrapping our analysis code into a class descended from Calvin data analysis abstract class, we do not have to change Calvin itself to explore different analysis methodologies. By replaying past data streams (in this case, sessions of web browsing), we can see how the analysis techniques are improving against a standard set of data. Because our current analysis techniques are stochastic, we also run the algorithm against the same data set multiple times to measure the variance in performance.

7. IMPLEMENTATION

Most of Calvin is written in Java, allowing it to be developed and tested in Windows, Unix, and Linux. The use of inheritance among the classes allows various components to be extended and recombined for different research tasks. The goal is to maximize the ability to reuse components so that when the research goals shift slightly, obsolete components can be deleted without requiring extensive rewrite of other components. This is achieved in part by abstracting both resource types and data analysis components. By defining abstract operations over resources, the specific kinds of resources used in Calvin can change without requiring Calvin itself to change. The primitive operations defined for a resource are shown in table 1. As described previously, new resource types are added to Calvin by writing a class which implements these operations for the new resource type. This class is added to the registry, effectively integrating the new type into Calvin without changing Calvin itself.

In order to abstract the data analysis component, we assume that every implementation of a data analysis component analyzes a stream

displayMe	Displays information about the resource
editMe	Displays a window for the user to edit the information
fetchMe	Load the resource into its native application
get/set Content	Returns or changes the full text of the resource
get/set IdString	Returns or changes the short natural language description of the resource
get/set Location	Returns or changes the location of the resource (such as URL, disk location, etc.
get/set Memo	Returns or changes user comments on a resource
get/set ResourceId	Returns or changes a unique internal identifier for the resource
get/set TypeId	Returns or changes the identifier for the type of resource this is.

Table 1: Primitive operations defined on Resources

closeContext	Stop using the current context
closeResources	Find resources sufficiently similar to the current context
contextId	Return the unique id for this context.
currentResources	Return the resources which have been accessed during this research session.
deleteResource	Remove all references to this resource from the database.
getControlPanel	Return a graphical control panel for users to change parameters and/or see the performance of the current context.
getUserList(Resource List)	Return a list of all users who have accessed this resource.
ignoreResource(resource id)	Do not count this resource as having been accessed during this research session.
initialize	Prepare the context for use for a specific research session.
initModel	Set up any underlying database structures necessary for this context model to work properly. Only can be done once per database.
registerHint	Tell the context model about a particular phrase that might be relevant to this context.
registerResource(resource)	Tell the data analysis component that a particular resource has been accessed.
retrieveResource(resource id)	Return the particular resource from the database.

Table 2: Primitives operations defined on contexts. As described in the text, "Context" in this table refers to a stream of data. A data analysis component processes the data stream to learn context-based indexing features for the documents the user accesses.

```
expla=# select pid as "USERNUM", avg(tfidf) as "TFIDF"
, avg(ws1) as "WS1" from simtocqone group by pid;
 USERNUM |        TFIDF        |         WS1
---------+---------------------+---------------------
       8 |   0.25044646091875  | 0.354190238911036
       9 | 0.264521606841747   | 0.215712595995764
      10 | 0.263711520765883   | 0.188364062124925
      12 |  0.26825657692134   | 0.356391264662916
      13 | 0.254392779750772   | 0.391180770295963
      15 | 0.316211565206512   | 0.409325107220054
      24 |   0.2657559200948   | 0.269685933338144
(7 rows)

expla=#
```

Figure 4: Postgresql Text Interface

of data, which we call a context. A data analysis component can then be defined abstractly as a set of operations over a context. See table 2 for the set of operations. By building classes that implement these operations, all the components of Calvin can use multiple, diverse versions without modification.

The only part of Calvin that is not written in Java is the Postgresql database used to store data for later retrieval and analysis. Postgresql is an open source object relational SQL database server. Calvin components communicate with it through a JDBC driver over a TCP/IP connection. Although this reduces Calvin's ability to be deployed on a wide scale for individual use, the SQL database makes it much easier to analyze and store large amounts of data. Data in Postgresql can be retrieved one of any number of ways. Postgresql's ODBC driver lets one import it into Microsoft Access or SPSS. The JDBC driver lets one access it via Java. Or one can use the command line sql utility included with Postgresql to access it directly.

Our infrastructure also includes utilities that process the data collected by Calvin to show how our information retrieval algorithm would have performed had the users been using an agent with some given data analysis component. Figure 4 shows the results of one such experiment, in which the average performance of the *term frequency/inverse document frequency* indexing algorithm compared to our current data analysis component. In this example, the data is grouped by user. But because of the infrastructure built into Calvin, with this data stored in the database, one could just as easily group by document, document length, or other attribute to perform any number of analyses.

8. TRADEOFFS

There is a tradeoff between flexibility and deployability in building research agents. Because Calvin stores its information in an SQL database back end, Calvin would be difficult to deploy to end users for wide scale testing. The database back end makes is easier to manipulate and analyze data in research situations, but it makes the software bulky for deployment. To make the system deployable, the agent would only store as much data as was needed, and use a simplified database which has only the needed features needed for the functioning of the chosen analysis engine.

Another tradeoff has to do with the capabilities of the agent toolkit. Flexibility was a design goal, but it is inevitable that constraints be put on the what kinds of information Calvin-based agents could perform. The Calvin toolkit is designed to analyze documents as the user accesses them. This is consistent with many kinds of existing personal information agents, but is inconsistent with supporting some approaches, such as the one taken by Rhodes' Remembrance Agent [11] (see section 11). Remembrance Agents suggest resources which are not necessarily accessed by the user, but have previously been indexed. Calvin's nature is to index documents as they are accessed, including documents such as WWW pages which are not on the user's hard drive.

Also, Calvin's multi-agent extensibility is purchased at some expense of complexity of use. Each agent has to be started individually because it is a separate application from any particular data analysis agent. However, this is an issue that can be dealt with by improving the XML messaging component of Calvin. With improved complexity in the kinds of messages Calvin can send, Calvin could start and control different agents from a common user interface.

9. FUTURE WORK

One area for improvement is the architecture's XML message passing. The message passing architecture is in place, but the vocabulary to which the toolkit is responsive could be improved to support two-way communication. Also, the ability for particular agents to extend the vocabulary of the XML Executor module would enhance the toolkit's usability. For example, currently the monitoring agents which send information to Calvin function as independent programs. If they could be made to respond to commands specified in XML such as "stop monitoring," or "return status," a control panel could be added to Calvin's user interfaces to control all the agents from one central panel.

Finally, a Java-based client database system would increase Calvin's distributability. Such a database system would relieve Calvin's reliance on a separate database server. This would increase the processing burden on the client workstations, but would provide a more compact software package. Existing products, such as InstantDB (http://instantdb.enhydra.org) do not have enough functionality to be useful yet for Calvin, although they are getting close.

10. AVAILABILITY

Many of the reusable utility classes we developed to build Calvin, along with other related classes useful in agent/AI research, have been released in the IGLU Java source code library, available from http://www.cs.indiana.edu/~trbauer/iglu. It is released under a FreeBSD-like license. We welcome suggestions and contributions.

Some of the other components used in Calvin are also freely available. The database back end, Postgresql is an open source project released under a FreeBSD-like license and is available for download at http://www.postgresql.org. Numerous XML parsers are freely available for Java. Calvin uses XMLtp, available from http://mitglied.tripod.de/xmltp. XMLtp is also released under a FreeBSD-like license.

The full Calvin toolkit has not been released. However, we welcome discussion on possible cooperative efforts to use and enhance the toolkit for greater use.

11. COMPARISON TO IR AGENT ARCHITECTURES

There are many other information retrieval agents which try to record, analyze, and respond to user behavior. The Watson project [2] analyzes the format of a document and relevant keywords are extracted to make a web search. The Watson project is not built as an application of a general agent toolkit, which would probably make it more difficult to be adapted to new kinds of IR agents. However, unlike Calvin, it is quite distributable because it requires no central database.

Remembrance Agents [11], which analyze existing documents on a person's hard drive and suggest them at appropriate times, use a common engine for analysis, so some of the existing analysis code is usable in multiple agents. Calvin, rather than focusing on reusable analysis code, focuses on reusable infrastructure into which different analysis algorithms can be run.

Some agents use an architecture which is similar to Calvin's [7, 6, 9]. These architectures use a proxy server to analyze pages being accessed by the user. The architectures focus more on observing WWW browsing activity than Calvin, which is built to support multiple data types. Unlike these architectures, Calvin has abstracted the notion of a resource so that resources of any type can be used.

Many other IR projects develop personal information agents to suggest resources to users. In the existing literature, however, the focus is often on the analysis techniques rather than on reusable architectures. Such agents [4, 1, 3] can observe document access for analysis. However, the focus on published literature of these agents is not reusability of the infrastructure, but the analysis of the documents.

Other software projects have gathered information about user activity, but not for the purpose of information retrieval. Gorniak and Poole [5], for instance, developed a system which records keystrokes in Java applications by forcing the applications to use a customized event queue in the Java machine. This kind of data collection technique is quite amenable to the Calvin architecture: A sub-agent could be implemented to gather this data and send it to the XML Executor.

12. CONCLUSION

Calvin is a research grade agent system which observes users accessing documents. It provides facilities for building agent that can index the documents, and make suggestions based on the documents accessed. The process of indexing and suggesting has been abstracted so that multiple data analysis components can be implemented without modifying the architecture. Interactions can automatically be recorded and "replayed," allowing the automatic creation of a test bed of data sets for determining the performance of analyzing algorithms. The kinds of resources the agents can recognize, as well as the ways in which the resources are accessed are extensible because of a built in resource registry and XML message passing over TCP/IP connections. Calvin is designed to help free the researchers from thinking about data storage/infrastructure issues, so that they can focus on the information analysis elements of the research.

13. REFERENCES

[1] E. Bloedorn, I. Mani, and T. R. MacMillan. Representational issues in machine learning of user profiles. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference*, 1996.

[2] J. Budzik, K. Hammond, and L. Birnbaum. Information access in context. In *Knowledge based systems*, 2001.

[3] J. R. Chen, N. Mathé, and S. Wolfe. Collaborative information agents on the world wide web. In *Proceedings of the third ACM Conference on Digital libraries*, pages 279–280, 1998.

[4] W. W. Cohen. Learning rules that classify e-mail. In *Papers from the AAAI Spring Symposium on Machine Learning in Information Access*, pages 18–25, 1996.

[5] P. Gorniak and D. Poole. Predicting future user actions by observing unmodified appliations. In *Proceedings of the 2000 National Conference on Artificial Intelligence*. AAAI, 2000.

[6] M. Jaczynski and B. Trousse. WWW assisted browsing by reusing past navigation of a group of users. In B. Smith and P. Cunningham, editors, *Advances in Case-Based Reasoning: 4th European worskhop*, Lecture Notes in Artificial Intelligence, pages 160–171, Doublin, Ireland, 1998. Springer-Verlag.

[7] T. Joachims, D. Freitag, and T. Mitchell. Webwatcher: A tour guide for the world wide web. In *Proceedings of IJCAI97*, August 1997.

[8] D. Leake, T. Bauer, A. Maguitman, and D. Wilson. Capture, storage and reuse of lessons about information resources: Supporting task-based information search. In *Proceedings of the AAAI-2000 Workshop on Intelligent Lessons Learned Systems*, Menlo Park, CA, 2000. AAAI Press.

[9] H. Lieberman. Letizia: An agent that assists web browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1995.

[10] J. Novak and D. Gowin. *Learning How to Learn*. Cambridge University Press, New York, 1984.

[11] B. J. Rhodes. Margin notes: Building a contextually aware associative memory. In *Proceedings of the 2000 international conference on Intelligent user interfaces*, pages 219–224, Jan 2000.