

Collapsible User Interfaces for Information Retrieval Agents

Martin Frank

Pedro Szekely

USC/ISI, 4676 Admiralty Way
Marina del Rey, CA 90292 USA
(310) 822-1511 ext. 182 or 641

martin.r.frank@acm.org or szekely@isi.edu

ABSTRACT

This paper presents an architecture for information retrieval agents in which each agent declaratively describes its domain, input, output, and user interface. A mediating piece of software can then assemble software agents for a given information retrieval task, and produce a single, unified user interface for that task from the individual agents' descriptions.

Keywords

Software agents, information retrieval, model-based user interfaces, graphical user interfaces, human-computer interaction

INTRODUCTION & MOTIVATION

The World-Wide Web now gives us ready access to many real-time price quotes. At the same time, many prices have become more volatile. For example, the price of an airline flight can change many times a day, and airlines increasingly demand immediate payment to secure a price quote rather than guaranteeing a reservation for several days. Given the above, the best strategy for buying an airline ticket is to constantly watch the fare for the particular date that you are interested in, using the Web site of the airline or of a travel agency (or ideally both), and to be ready to immediately purchase the ticket if the fare drops.

Manual tracking would quickly become tedious. At the same time, supplier-side notification services ("give us your email address and we'll keep you posted") are not always available, often don't let you specify precisely what you are interested in, and can't generally be trusted to act in the best interest of the consumer. For those reasons, automated tracking presents a prime opportunity for software agents. In the airline domain, what one would want is automatic charting of past prices (so that one has an idea of what a "good price" is) as well as a notification when the price changes (so that one can react immediately).

Appearing in:

1999 International Conference on Intelligent User Interfaces
Redondo Beach (Los Angeles, California USA)

January 5-8, 1999

--- COPYRIGHT ACM 1999 ---



Figure 1: How the agents should *not* appear to the user

One possible approach to building such a system is to construct a monolithic application specialized for the airline domain. However, that would preclude easily re-using common components such as data tabulation and presentation for similar domains (say car rental reservations or real estate market tracking), and it would not let one plug in new capabilities on the fly (such as adding a new data source or a new graphing capability).

We have therefore taken a different approach in which several software agents work together in retrieving data from several sources, tabling it, and producing Web pages for the end user. Each declaratively describes its input, output, and parameter requirements. A mediating piece of software called the Scheduler then chains the agents together in a way that "makes sense", and collapses (blends, folds) their user interfaces into one. The prime design goal was that the agents should appear as individual entities from a software engineering perspective (maintainability, scalability) yet appear as a unified system from a user

interface perspective (usability).

Figure 1 shows an example of what we did *not* want the user interface to multiple agents to look like – the agents appear as separate entities that are given instructions independently. (The snapshot is adapted from an early version of our system that we built to exercise and debug the individual agents.)

WALK-THROUGH: TRACKING REAL ESTATE

This section describes the use of the Agent Playground from an end-user perspective. Imagine that you are about to buy a home in the Los Angeles area and want to keep track of home prices in neighborhoods you are interested in.



Figure 2: Walkthrough: Welcome Screen

Figure 3 shows the screen after the user clicked on “new inquiry” for the real estate domain. (Ignore the fact that an inquiry for El Segundo already existed in Figure 2 - we are just re-tracing the steps of the user here for the sake of this walkthrough.) The HyperText Markup Language (HTML) of this page is not hard-coded – it is synthesized on the fly based on the user interface needs description of the available agents in that domain.

Figure 4 shows the result summary for our “El Segundo” inquiry, which first lists the output of the domain-specific agents, followed by a list of links to Web pages produced by automatically scheduled post-processing agents. (We are currently working on improving the readability of the results by better modeling agent output.)



Figure 4: Walkthrough: Result Overview

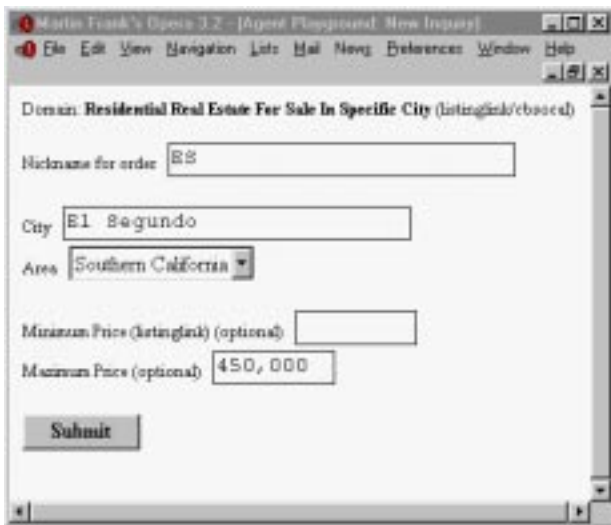


Figure 3: Walkthrough: Entering a new inquiry

Figure 2 shows the initial screen, now based on application domains rather than individual agents.

After receiving a new inquiry, the system automatically schedules times at which the real estate agents will fetch new data (also based on the agents’ descriptions as explained later, two times a week in this particular example). An inquiry can also be executed manually via the “run now” links in Figure 2.

Finally, Figure 5 shows the most interesting result data for real estate inquiries. The first two columns of Figure 5 contain the dates between which the change occurred, the third column describes the nature of the change, and the remaining columns reflect the data from the original tables. If the change was a modification, an arrow (“->”) points out the old and new values of the column.

The table was produced by a post-processing agent which tracks the changes to a table from run to run; this agent is automatically scheduled for any data-gathering agent that produces a table. In the real estate domain, this results in a compact summary of real estate market developments. For example, a prospective buyer could determine the prior history (changes in listing price, time on the market, etc.) of a property by searching for the street address in this table.

Start Date	End Date	Status	Price	Bedrooms	Bathrooms	Property Type	Other	Address	City	Agent
Mar. 3, 1998	Mar. 5, 1998	added	\$352,500	3	1.75	Fp		1867476 Address Withheld	El Segundo	Shorewood Realtors
Mar. 5, 1998	Mar. 12, 1998	deleted	\$259,000	3	1.75			1770537 124 W Walnut Av	El Segundo	RE/MAX Beach City-El Segundo
Mar. 5, 1998	Mar. 12, 1998	added	\$359,000	4	1.75	Fp		1951705 227 E Walnut Av	El Segundo	Schofield Realty
Mar. 12, 1998	Mar. 17, 1998	added	\$415,000	3	2.5	Fp	P	1790927 1204 E Maple Av	El Segundo	Shorewood Realtors
Mar. 17, 1998	Mar. 19, 1998	modified	\$359,000 -> \$337,000	4	1.75	Fp		1951705 227 E Walnut Av	El Segundo	Schofield Realty
Mar. 26, 1998	Mar. 31, 1998	deleted	\$299,000	3	2	Fp	S A P	1938394 901 Center St	El Segundo	Real Estate West
Mar. 31, 1998	Apr. 2, 1998	modified	\$415,000 -> \$409,000	3	2.5	Fp	P	1790927 1204 E Maple Av	El Segundo	Shorewood Realtors -> Lynn C. O'Neil

Figure 5: Walkthrough: Partial results for our running example

(Disclaimer: We are investigating auto-tracking of commercial web sites' content for academic and personal use only. Any resale or republishing of such data would likely require a license.)

THE AGENT MODELLING LANGUAGE

Declarative agent descriptions are central to synthesizing user input forms on the fly. We will present them here, and subsequently discuss the algorithms operating on them. We have slightly prettified our actual ASCII-based, LISP-ish looking representation for human readability.

Airline Reservation Agents

In addition to the real estate domain, tracking airline fares is another application of the Agent Playground.

The ITN Agent

This section describes the model of the agent that retrieves flight price information from the Web site of the Internet Travel Network (ITN).

Like all our domain-specific agents, it declares which parameters it needs from the user for its query to the Web site. When it is executed, it will first go to the Web site's home page, then possibly follow a number of links and programmatically fill out a number of HTML forms, extracting results along the way by parsing the HTML pages it retrieves in a heuristic manner. It then writes the results to a file in a standard attribute-value format (FlightTime "4:10 hours", RoundTripPrice "\$459.00"). These results can then later be post-processed by other agents.

```
Agent itn "ITN Airline Agent"
Domain roundtripAirlineTicketInfo
Description "Lets you determine the ..."
```

The agent's name and description can be used by the Scheduler to present it to the user; "itn" is the internally used short name for the agent. The domain identifies the applicability of the agent (based on simple string matching with the domains of other agents, there is no ontology of agent domains yet).

```
Par apDir
```

Agent parameters starting with lower-case "ap" (the initials of Agent Playground) followed by an upper-case letters are provided by the system whenever an agent is run. Since the semantics of those parameters are built into the system there is no need to elaborate on them in the agent description. Above, the ITN agent indicates that it needs the name of a directory in which it will write its output files. The other current system parameter is *apUser*, which is requested by post-processing agents because it can uniquely identify another job (in combination with an agent name and a job name). We will omit system parameter requests in the remaining agent descriptions.

```
Par itnLogin(
  description="Your ITN login name",
  doNotEcho,takeFromPreferences)
```

The description can be used to prompt the user; we have deleted the description string of all further parameters in this paper for brevity. The caller is instructed not to echo the value of the parameter to the output it produces, and that it should take the value of this parameter from user-specific preferences. Finally, the parameter has no type, meaning that the system should pass the user-provided text to the agent as-is.

```
Par itnPassword(
```

```

doNotEcho, takeFromPreferences)
Par origin(type=AirportCode beginGroup)
Par destination(type=AirportCode endGroup)
Par initialDepartureMonth(
  type=Month("Jan")
  beginGroup("Initial Departure"))
Par initialDepartureDay(type=Day("1"))
Par initialDepartureTime(
  type=Hour("12 am/6 am/12 pm")
  endGroup)
Par returnDepartureMonth(type=Month("Jan")
  beginGroup("Return Departure"))
Par returnDepartureDay(type=Day("1"))
Par returnDepartureTime(type=Hour("12 am")
  endGroup)
Par airline(
  type=Airline("American")
  legacySynonyms(initialAL, returnAL))

```

The type of the parameter above is Airline, with a type format addendum that requests the airline's name - not symbol - be passed (e.g. "United", not "UA"). There is a finite list of types that are currently recognized by our system, we do not yet have constructable types - if a new agent needs a type outside of the list, the procedural Java code implementing the type ontology must be extended.

The list of legacy synonyms of the *airline* parameter indicates that the agent in the past used different names for this parameter (this is valuable because the system can then compensate for name changes when it tabulates output over time).

The origin and destination parameters above are declared to be an (unnamed) group. Similarly, initialDepartureMonth, initialDepartureDay, and initialDepartureTime logically belong together (making up a group named "Initial Departure").

```

Par initialFlightNo(
  type=AirlineFlightNumber beginGroup)
Par returnFlightNo(
  type=AirlineFlightNumber endGroup)
Par keepAllPages(
  type=Boolean("yes/no"),
  default="no", optional)
Input website("www.itn.com")
Scheduling scheduleAtTimeAndIncrement(
  "H 7 Min 30", "Min 5")

```

Each agent describes where its input comes from in a rudimentary form. For example, the input information above is used by the garbage collection agent, which will refuse to delete output from agents that produce "original" output (defined to be those that take their input from external Web sites, rather than from files on disk).

The scheduling information translates to "schedule jobs for this agent at 7:30am, or in five minute increments after that time if a job already exists for the previous times". Thus, jobs can be scheduled for a time that is appropriate for the agents (we may know that a Web site's data tends to get updated at 7am every morning, or that it changes only twice a week), the increment exists to avoid putting a heavy load on the Web site through a large number of automated requests.

Just as the agents describe their input parameters they also describe the attributes of their output (such FlightTime and RoundTripPrice). However, we will omit our representation for describing agent output attributes here because we are still working on the algorithms making use of them.

The Delta Agent

The description of the Delta agent is similar to that of the ITN agent, so that we will only describe differences here.

Most important is the implicit parameter for the airline of value "Delta" (an airline's web site will typically not offer tickets for its competitors). There are also subtle differences in the time input format that the Delta agent expects (e.g. "May 07, 12 Noon", not "May 7, 12 pm).

```

Agent delta "Delta Agent"
Domain roundtripAirlineTicketInfo
Description "Retrieves flight price ..."
Par origin(type=AirportCode beginGroup)
Par destination(type=AirportCode endGroup)
Par initialDepartureMonth(type=Month("Jan")
  beginGroup("Initial Departure"))
Par initialDepartureDay(type=Day("01"))
Par initialDepartureTime(
  type=Hour("12 Midnight/6 AM/12 Noon")
  endGroup)
Par returnDepartureMonth(type=Month("Jan")
  beginGroup("Return Departure"))
Par returnDepartureDay(type=Day("01"))
Par returnDepartureTime(
  type=Hour("12 Midnight/6 AM/12 Noon")
  endGroup)
ImplicitPar airline(
  type=Airline, value="Delta")
Par initialFlightNo(
  type=AirlineFlightNumber beginGroup)
Par returnFlightNo(
  type=AirlineFlightNumber endGroup)
Par restrictedFare(
  type=Boolean("yes/no"), default="yes")
Par noOfPassengers(
  type=RangedInteger(1,4), default="1")
Par keepAllPages(
  type=Boolean("yes/no"), default="no",
  optional, doNotEcho)
Input website("www.delta-air.com")
Scheduling scheduleAtTimeAndIncrement(
  "H 7", "Min 10")

```

Real Estate Agents

As described in the Introduction, we have also written agents that automatically retrieve real estate information from the Web.

The ListingLink Agent

ListingLink is probably the most comprehensive real estate Web site at the moment. Our ListingLink agent retrieves the homes currently on the market for a given area and city by simulating a user interacting with that Web site. The real estate agents each produce a whole table (of the homes currently available), not just attribute-value pairs as the airline agents do. For that reason, they explicitly describe their output in an Output parameter (describing the name and type of the file, but not yet the semantic contents of the file).

```

Agent listinglink "ListingLink Agent"

```

```

Domain residentialRealEstate-
  ForSaleInSpecificCity
Description "Gives you a table ..."
Par city(type=String(averageLength=20))
Par area(type=RealEstateArea
  legacySynonyms(state))
Par minimumPrice(
  type=String(averageLength=8)
  optional)
Par maximumPrice(
  type=String(averageLength=8)
  optional)
Input website("listinglink.com")
Output singleTypedOutputFile(
  tabular,"table.native")
Scheduling scheduleAtTimeAndIncrement(
  "D Tue,Thu H 20","Min 10")

```

The Southern California Coldwell-Banker Agent

This Web site offers the same type of information, but sometimes lists some properties that are not found in Multiple Listing Service sites such as ListingLink.

```

Agent cbsocal "Coldwell Banker Southern
  California Agent"
Domain residentialRealEstate-
  ForSaleInSpecificCity
Description "Gives you a table of..."
Par city(type=String(averageLength=20))
ImplicitPar area(type=RealEstateArea,
  value="Southern California")
Par maximumPrice(
  type=String(averageLength=10)
  optional)
Input website("www.cbsocal.com")
Output singleTypedOutputFile(
  tabular,"table.native")
Scheduling scheduleAtTimeAndIncrement(
  "D Tue,Thu H 20 Min 5","Min 10")

```

Domain-Independent Post-Processing Agents

In addition to the “wrapper” agents for particular web sites we also wrote a number of post-processing agents which tabulate and present data from the wrapper agents. This tabulation could occur on user demand, or could be pre-processed every time new data is produced. We have take the latter approach because the data sets are often large so that post-processing can sometimes take 2-3 minutes. In addition, our architecture allows other post-processing agents to be added later which will then automatically be scheduled

The Tabler Agent

The *tabler* agent puts the data that other agents have produced over time in a tabular format. For example, imagine that it post-processes the attribute-value output from a simplistic car rental agent that just writes out the time it ran and how much the specified car rental would cost, and that this agent job was run 12 times. The *tabler* would then gather the attribute-value pairs for each run from disk, and produce a table two columns wide and twelve rows high (with the columns containing the values for the date and the price).

The empty *Domain* field below indicates that it is always

applicable. The *Scheduling* field indicates that the agent should be scheduled after every job of an agent that gathers input from an external web site, and that the *agent* and *order* parameters should be filled in with the information that identifies the agent whose data we are tabling. No *Input* description is necessary because the input of the agent is implied in the *Scheduling* information.

```

Agent tabler "Tabler Agent"
Domain
Description "Makes a table out of ..."
Par agent(type=AgentShortName)
Par order(type=JobId("19961224173000"))
Output singleTypedOutputFile(
  tabular,"\table.native\""),
Scheduling scheduleForEveryAgentJob(
  agent,order) agentMustBeOriginal()),

```

The Table-Tracker Agent

Some original agents such as the real estate not only produce single attributes-value pairs as output (such as the number of listings retrieved) but also additionally write whole tables of information (such as the listing of homes on the market). For these agents, the Table Tracker can compute a table that summarizes the history of changes in a time series of tables. This resulting table of changes lists two points in time between which the change occurred, then lists the type of change, and then the entries from the original tables (see Figure 5 for an example).

```

Agent tabletracker "Table Tracker"
Domain
Description "Summarizes the changes to ..."
Par agent(type=AgentShortName)
Par job(type=JobId("19961224173000"))
Par inputFile(type=RelativeFileName
  optional default("table.native"))
Output singleTypedOutputFile(
  tabular,"\table.native\"")
Scheduling scheduleForEveryFileOfType(
  tabular,agent,job,inputFile)
agentMustBeOriginal()

```

The Web-Page Producer

This agent takes a table in our native format and produces an HTML version of it.

```

Agent table2html "Table To HTML"
Domain
Description "Produces a nice-looking ..."
Par a(type=Agent)
Par j(type=JobId("19961224173000"))
Par f(type=RelativeFileName
  optional default("table.native"))
Output singleTypedOutputFile(
  html,"table.html")
Scheduling scheduleForEveryFileOfType(
  tabular,a,j,f)

```

DECIDING WHICH AGENTS SHOULD BE INVOLVED FOR A DOMAIN

There are two stages to generating a domain-specific inquiry entry form. This section describes the first step, which consists of deciding which agents should be involved, in what order, and when they should be run. The next section will then describe the second step of generating a user interface once it is known which agents are involved.

We named the sub-system for the first step the Scheduler. Given a domain identifier (a simple string) it will first gather all agents in that domain, and schedule them for the time that each asked for in its *Scheduling* field. If that time slot is already taken by a run of the same agent, it will be scheduled for the first available time slot, based on the increment given in the *Scheduling* field.

The Scheduler will then recursively add post-processing agents to the original domain agents until no more post-processing agents are applicable. The current post-processing agents are of two categories: agents that apply after each original agent, and agents that apply after others that write files of a certain type (the latter may also be restricted to original agents, or may not be). The *tabler* falls into the former category because there are always some output attributes for any agent run – even if an agent does not produce any output, the Daemon running it will add some standard output attributes such as when the agent was run, and how long it took to execute it.

Each post-processing agent will point out the parameters it expects to be filled in, which identify on what data it will run (consisting of an agent short name, a job id, and possibly the name of the file that they are post-processing). The Scheduler adds this information as it constructs the agent chain.

In the airline domain, this process results in the agent order shown in Figure 6. We are currently working on modeling agent output in enough detail so that we can also combine the *output* from several different original agents in a meaningful way. For now, the Agent Playground does not combine data from different agents.

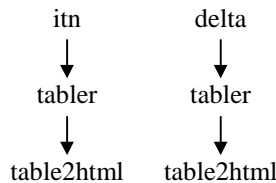


Figure 6: Resulting Agent Chain, Airline Domain.

Figure 7 shows the chain of agents for the real estate domain.

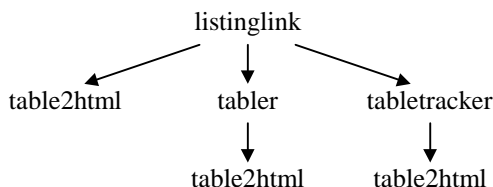


Figure 7: Resulting Agent Chain, Real Estate Domain.

The *tabler* is scheduled after *listinglink* because it produces attribute-value data. *tabletracker* and the immediate *table2html* run after *listinglink* are scheduled because it

produces a tabular data file. From left to right, the resulting three tables answer the following questions. (We have omitted an equivalent graph for the *chsocal* agent.)

- At what times did *listinglink* run, how long did each run take, and how many homes overall were on the market when the agent ran?
- What were the changes in the market over time (as in Figure 5)?
- What is the most recently retrieved list of homes currently on the market?

GENERATING THE DOMAIN-SPECIFIC END-USER INTERFACE

The user interface for giving instructions to the agents within one domain is synthesized based on the agent descriptions, an ontology of the parameter types requested by the agents, and a storage facility for parameters that are applicable to more than one inquiry. We will first discuss the latter two sub-components, and then describe the user interface generation process.

The Agent Parameter Ontology

We maintain a central ontology of parameter types that is used to check the validity of given types and to translate between different formats for these types.

Both the scope and the implementation of this ontology are primitive. The types covered by the ontology are only those that we needed for the airline domain, the car rental domain, and the real estate domain (the domains overlap, of course, especially the airline and car rental domains).

The questions that can be asked of the ontology are the following:

- Is the given string a valid parameter type?
- Do the following two strings represent the same value for a given type?
- For a given type (“Day”), type format identifier (“01”), and value (“7”), what is the correct representation for that value? (The answer is “07”.)
- For a given enumerated type, what are all possible values, in the canonical type format? For example, the values of type Month are “Jan”.. “Dec”.
- For a given textual type, what is the average number of characters needed? (An AirportCode is 3 characters long.)

The Preferences Repository

The Scheduler also maintains a file of user preferences in simple attribute-value format. If an agent declares that a certain parameter should be taken from the preferences (such as *itnLogin* and *itnPassword* for the ITN agent), the Scheduler will take its value from there. If it is not found, it will appear as a regular-entry field this time, and be put in the preferences file for future use when the user submits the form.

The User Interface Generation Algorithm

Based on the order of agents applicable to this domain computed in Step 1, we can obtain a list of parameters needed. Some of these parameters will have already been filled in by the Scheduler in Step 1, such as the parameters identifying the preceding job for post-processing agents. Some others are supplied by the Daemon whenever a job is run, such as the directory to write output to.

We group the remaining parameters into required and optional ones, retaining the order of parameters listed by the first agent encountered. Parameters of subsequent agents are added at the end of the list (unless their name and type matches an existing parameter, in which case they are not added at all).

From that list of user parameters needed, we remove those that indicate that they should be taken from the preferences (assuming the preferences actually contain a value for them).



Figure 8: Generated user interface for the airline domain

For any remaining parameter, we query the parameter ontology for a possible default value, plus either for all its possible values (enumerated types) or for its textual length (non-enumerated ones). Finally we put parameters bracketed by the *beginGroup* and *endGroup* constructs in the same line. If the *beginGroup-endGroup* construct provides a name for the group we will omit individual field names and use the group name as a label describing all of them.

Figure 8 shows the user filling out the form that is automatically generated for the airline domain. For example, the “Initial Departure” line consists of a named

group, the “Initial Flight No.” line is an unnamed group. For parameters not requested by all agents involved, we list the short names of the agents asking for them (as in the “Restricted Fare” line).

Figure 3 in the Introduction is a similar, simpler example of an inquiry entry form that was generated in the same way.

IMPLEMENTATION

The Agent Playground is a Web site based on Java servlets. It can be used with any browser that supports HTML forms and tables. A “daemon” Java process runs in the background, and wakes up every five minutes in order to check the running instructions of each user inquiry and executes the inquiries if they became due.

We are currently in the process of converting the user input forms (Figures 3 and 8) from simple HTML forms to our Adaptive Forms [4], which have the advantage that they can encode some dependencies between fields.

RELATED WORK

Our research is most related to “model-based” user interfaces, which do not hard-code their graphical user interfaces but instead assemble them on the fly from declarative specifications of what is to be presented to the user [3,6,7]. Our user interfaces only consist of simple forms, on the other hand we try to infer not only the user interface from the specifications but in a limited sense also assemble the application itself on the fly (which agents should be involved, and in which order).

On the software agents side, our work is related to automated information retrieval agents [2,5]. Our focus is different in that our agents themselves are deterministic and distinctly un-intelligent; our emphasis is instead on intelligence in the user interface.

One particularly interesting information retrieval agent is ShopBot [1] which is a software robot that given a list of on-line vendors’ Web sites and a set of product descriptions to search for can learn how to query those sites for price information. This is done by first looking for pages that resemble a search form, and by then repeatedly filling in permutations of the given product description into the search form until the result page looks like a product description for the product examples. Once ShopBot has learned how to find and fill in the search form for a particular vendor, it can retrieve product information for that vendor instantly, and end-users can issue simultaneous queries against multiple vendors. ShopBot is superior to our individual agents in the sense that its wrapping of sources is more robust (less likely to break if Web sites are re-designed). We believe our system is of value if the user explicitly wants to track sources over time (ShopBot answers questions of the kind “what is available right now”), or in domains where there are few Web sources of interest (so that writing agents to retrieve information from each manually is actually simpler and faster than modeling the domain in such detail so that a generic software robot

could retrieve the information).

FUTURE WORK

There are numerous shortcomings in our current implementation. For example, we should use an existing AI planner rather than our ad-hoc scheduling algorithm, and knowledge about parameter types should be stored declaratively.

We are currently working on better modeling the output of agents so that we can present results by their semantics rather than by the agent chain that produced them (as was done in Figure 4). In addition, more detailed modeling will also enable combining the output from several parallel agents, so that e.g. the query results for several airline agents can be summarized in a single table (rather than in one table per agent as we do now).

We also want to add a post-processing agent that sends email when specified changes occur. The input parameters to this agent will depend on the output of previous agents. For example, the user should be able to select "price" as an attribute of interest only if the preceding agents actually produce such an attribute. This should also be possible with more detailed agent output modeling.

CONCLUSION

The Agent Playground presents a very small step towards a full-fledged application that can dynamically adapt its user interface to new functionality added on the fly. We nevertheless believe that architectures in which agents describe their user interface needs declaratively represent a promising new research direction.

ACKNOWLEDGMENTS

We gratefully acknowledge DARPA funding of the MasterMind project as part of the Human-Computer Interaction Initiative and of the JFACC program, and thank Hans Chalupsky and Devang Patel for their suggestions.

REFERENCES

1. R. B. Doorenbos, O. Etzioni, , and D. S. Weld. A scalable comparison-shopping agent for the world-wide web. In Proceedings of the First International Conference on Autonomous Agents, pages 39-48, (Orlando, Florida, January 6-9) 1997.
2. O. Etzioni and D. Weld. A softbot-based interface to the Internet. Communications of the ACM, 37(7):72-76, July 1994.
3. J. D. Foley, W. C. Kim, S. Kovacevic, and K. Murray. Defining user interfaces at a high level of abstraction. IEEE Software, 6(1):25-32, January 1989.
4. M. Frank and P. Szekely. Adaptive Forms: An interaction paradigm for entering structured data. In Proceedings of the ACM International Conference on Intelligent User Interfaces, pages 153-160, (San Francisco, California, January 6-9) 1998.
5. S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based web agents. In Proceedings of the First International Conference on Autonomous Agents, pages 59-66, (Orlando, Florida, January 6-9) 1997.
6. P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools: the Mastermind approach. In L. Bass and C. Unger, editors, Engineering for Human-Computer Interaction. Chapman & Hall, 1996.
7. S. Wilson and P. Johnson. Empowering users in a task-based approach to design. In Proceedings of the ACM Symposium on Designing Interactive Systems, pages 25-31, (Ann Arbor, Michigan, August 23-25) 1995.