

A Modified Architecture for Constructing Real-Time Information Gathering Agents

Thomas Wagner John Phelps Yuhui Qian Erik Albert
Glen Beane

MaineSAIL (Software-agents and AI Lab)
University of Maine
wagner@umcs.maine.edu

Appears in the Proceedings of Agent-Oriented Information Systems, 2001

Abstract

Based on experiences learned from the BIG [9, 16, 17] information gathering agent we have generated a new approach to sophisticated information gathering agent construction. The new architecture removes some research-grade AI components, replaces others, and reframes the problem space to support stronger domain problem solving. The end result is an architecture that will support crisis response information gathering, travel planning, and music-centered digital library information gathering. In this paper we discuss the new architecture, provide rationale for change and restructuring, and identify the new technologies. Examples are framed in the travel planning domain as this application is the closest to actual deployment. The advantage of using agent technology in these applications is that it enables the software to meet real-time deadlines, respond to the dynamics of the Internet environment, reason about client preference, and plan to achieve the objectives. An agent approach will also facilitate future work in distributing the computation to multiple agents.

1 Introduction

The web is a rich and fertile information resource. It is also a time consuming and complex resource to use, for humans and software agents alike. Web-based information is generally unstructured or semi-structured, has varying degrees of quality, and is often contradictory. The larger Internet environment is similarly unruly – sites may be up or down, network bandwidth may vary, remote queries may take predictably longer at certain times of the day or response times may spike in a seemingly random fashion. While advancing technologies, e.g., XML, may alleviate some of the problems of working with the web, we do not foresee it becoming a globally cohesive infor-

mation resource at its base level in the near term – due to its size, highly distributed and individual nature, and due to the intellectual property issues involved.

In our research, we attempt to address the complex dynamics of the web as it exists today, and to do so in such a way as to leverage likely near-term advances such as the widespread use of XML. Our research builds on the hypothesis that much web-related information gathering is done to support a decision process, e.g., which product to buy or whether it is safe to travel to some distant country. Our overall objective is to enable a human user to describe the inputs to his or her decision process and from said specification to automate the gathering, extraction, and assimilation of information, and to perform some lower level decision functions, e.g., *suggesting* which product to purchase or *pruning* products that clearly do not meet the specification. This is a specific class of information gathering activity and we believe that for many such applications on-line information gathering is required because the information need is highly customized and situated in time, i.e., the more up to date information we have, the better.

Implementationally, the agent components and architecture described in this paper are being used to support crisis response planning (e.g., chemical spill response), travel planning, and music-centered digital library efforts. In this paper we focus primarily on the travel planning domain because it is the closest of these efforts to actual deployment. The architecture of this work is derived from our previous work in sophisticated problem solving agents in general, e.g., [15], and the BIG [9, 16, 17] information gathering agent in particular. BIG (resource-Bounded Information Gathering) is an agent that can, with little information *a priori*, recommend software products to users by performing online search and discovery, gathering, and processing of free format information. BIG uses a large set of research-grade AI technologies, e.g.,

blackboard problem solving [3, 4], information extraction [21], and Design-to-Criteria scheduling [26, 25, 24] for soft real-time agent control¹, and the integration of these technologies is an interesting topic in and of itself. BIG is an ambitious research artifact. However, BIG’s complexity is an obstacle when it comes to actual online use in real application settings. Components like the blackboard problem solver and the MUC-style (message understanding conference) information extractors are 800lb problem solving gorilla’s that may not be needed for certain applications where the problem structure is well defined and predictable results are more important than attempting active-search-and-discovery online.

Based on experiences learned from the BIG project, we have generated a modified approach to sophisticated information gathering agent construction. The objectives of the revised architecture are: 1) to improve the understandability and predictability of the information agent(s), 2) to simplify the process of re-deploying the artifacts in different application domains, 3) to facilitate a multi-agent approach to problem solving as in the WARREN portfolio management system [6], and 4) to enable others to more easily reproduce the agent or to build on the work. The new architecture removes some AI components, replaces others, and re-frames the problem space to support stronger domain problem solving. While we have removed many components from BIG’s prototypical architecture, the most notable are the blackboard planner/problem solver, the MUC class of information extraction technologies, and the document classifier required to keep BIG from getting distracted (and recommending Adobe Acrobat as the word processor of choice) [16]. We have also rebuilt all of the new agent components from scratch with the exception of the Design-to-Criteria scheduler – notes on the implementation can be found in Section 4. New technologies being deployed in the current architecture include an information gathering planner and a decision making component that is active in the assimilation of information. The architecture and these components are described in greater detail in the following sections.

It is important to note that the application domains targeted by this research are more focused and limited than those targeted by the BIG project. BIG is designed with the underlying objective of creating an approach to agent information gathering (for decision support) that is general, flexible, and employs powerful technologies for reasoning about information and extracted information. For example, BIG associates certainties with information extracted from documents and attempts to resolve conflicting data or uncertain data. This issues arise be-

¹Soft real-time is used here to denote a mix of real-time in the hard-real time sense and “fast enough for the application” real-time which is the way the expression is typically used in AI.

cause BIG attempts to support gathering from arbitrary sites and processing of information in arbitrary formats in much the same way that a human user would. In the current line of research, we have replaced arbitrary online search, discovery, and processing with a predefined library of online resources to which an agent may go for information. The agent also has a customized extractor for each known site – an approach typically used in information gathering agents, e.g., [8].² The implications of this are that our agents do not attempt to locate new information resources online while responding to a user’s query, thus, the agents are more predictable but also less able to adapt. For example, if all known sites are down, our agents cannot simply begin searching via AltaVista for new resources. However, in the BIG project, it became clear that arbitrary search and discovery is a very hard problem that requires all of BIG’s heavyweight problem solving, e.g., associating certainties with data extracted from web documents. This is why the activity of learning to use a resource and using the resource are often split as in [8]. The modified architecture also differs in that we have replaced the complex fusion performed by BIG’s blackboard with the activity of explicitly modeling the information produced/required by individual actions and by a decision making component that reasons about information assembly (where the information quality and type is highly predictable). In a way, explicit planning plus custom information extractors replace BIG’s blackboard problem solving and general extractors to produce more deterministic agent behaviors. Primary similarities between BIG and the current work are that agents reason about time limitations and resource bounds to produce results in soft real-time and that both attempt to address the “entire picture” to produce usable results. In the following sections we shift gears and focus on an instantiation of our modified agent architecture. Research issues and components are thus discussed in the context of the trip-planning application.

2 Related Work

This research falls into the general category of “moving up the food chain” [10] and constructing stronger information support than is provided by pure information retrieval technologies [2, 14]. In most higher level work, the objective is to return something other than a list of URLs to the users. Examples of this include personal shopping agents that perform price comparisons such as the original BargainFinder [13] and the ShopBot [8], though other types of personal agents abound [1, 20]. Some of the major differences between the work presented here

²When XML becomes prevalent, the custom extractors can simply be replaced.

and other personal information agents are that our agent reasons explicitly about time and resource limitations and that our agent performs more complex assembly of extracted information. For example, BargainFinder essentially returns prices whereas the agent presented here reasons about gathered information and preference and returns completed itineraries. A more complete examination of BIG and our agent versus other technologies can be found in [17].

3 Tripbot: An Architecture Instantiation

Along with the exponential increase of various online information resources, there are more and more online services available related to the travel-planning domain. For example, Travelocity (travelocity.com), Expedia (expedia.com), Internet Travel Network (itn.com), and TravelWeb (travelWeb.com) etc. are all popular sites for travel services, such as air flights, hotels and rental cars. When planning a trip online, a traveler often requires a number of services that are closely interrelated. Although each service may have individual constraints, the whole itinerary is also likely to be constrained by some overall factors such as activity scheduling or total budgets. However, most travel service sites require the user to schedule different types of services individually rather than consider them together as a complete itinerary, and most of them do not support tradeoff analysis. Therefore, the users have to assemble these services together by themselves and to make sure all the services are compatible and satisfy all individual and overall constraints. For a longer trip involving many activities, there are generally too many alternatives and constraints for simple and fast evaluation by humans – which makes trip planning a tedious and time-consuming task. Our solution is to construct an autonomous trip planning agent, called TripBot, to help people reduce information overload by carrying out certain classes of travel planning and itinerary construction automatically.

3.1 Architecture Overview

Figure 1 shows the prototypical architecture as instantiated for the TripBot travel planning agent. Control flow begins and ends with the GUI at the left side of the figure. The key components, in control flow order, are:

GUI The web based user interface is where the travel planning process begins. Via the interface users express trip objective criteria, e.g., travel dates, cost limits, and desired vacation classes (beachy versus

mountains). Users also specify preferences for different forms of entertainment, e.g., concerts, plays, golf courses, and specify preferences for hotel types and rental cars classes. Input also includes the desired time for the agent to search, e.g., “spend no more than 5 minutes planning my trip.” This input forms hard and soft constraints on the agent’s information gathering process and on the way it chooses between possible itinerary components.

Query Processor Query data is generated by the GUI and passed to the query processor which in turn expands certain keywords using a semantically organized lexicon. For example, if a user includes “hiking” as a keyword in the recreation category, then a closely synonymous activity, if available in the lexicon, such as “rock-climbing” could be added to the query, with lower priority than primary keywords. This enables the agent to search for related information when it uses keyword driven, site-specific, search engines.

Capability Assessor The raw query data plus the supplementary data produced by the query processor is passed to the capability assessor component. This component is a planner that plans to satisfy the query. The planner, described in greater detail in Section 3.3, plans from first principals but instead of producing a fully ordered sequence of actions, or a partially ordered sequence of actions, the planner enumerates a subset of the agent’s problem solving options and describes them statistically in terms of quality, cost, and duration. Instead of emitting a single plan, the planner emits a family of plans modeled in the TÆMS task modeling language [7, 23], which is the modeling framework used to represent and reason about the agent’s problem solving options. This is discussed further in the context of the Design-to-Criteria scheduler below. Along with the family of plans, the planner also emits a dependency specification that defines the expected inputs and outputs of actions as they execute. TÆMS is deliberately abstract and lacks such notions. Both planning from first principles and the typed dependency based reasoning are features unique to our new architecture and not found in BIG.

Design-to-Criteria Scheduler The TÆMS task structure emitted by the capability assessor (above) is then passed to the Design-to-Criteria scheduler for analysis. The scheduler has been used in numerous agent projects to evaluate the agent’s options and to determine a course of action for the agent, e.g., [15]. The scheduler is documented independently in [26, 25, 24]. The scheduler is part planner and

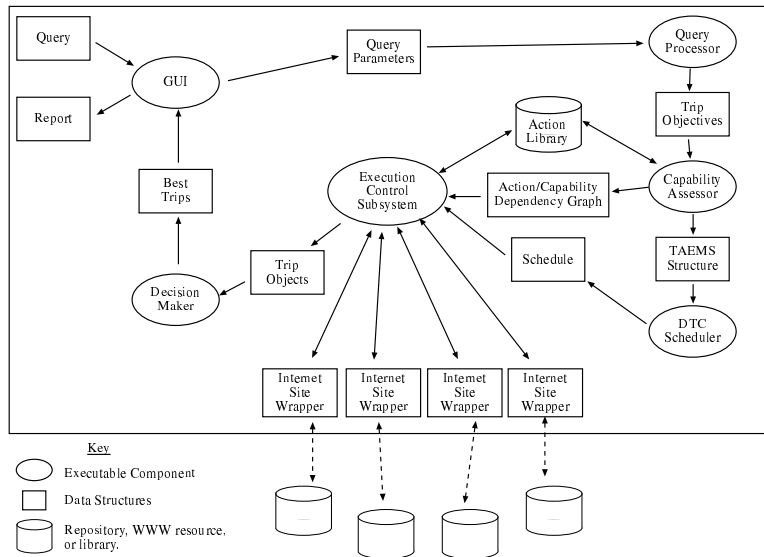


Figure 1: The TripBot Agent Architecture

part scheduler in that it decides which tasks to perform and how to go about performing them in addition to sequencing out the agent's activities. The scheduler reasons about deadlines, resource limitations, abstract interactions between tasks, cost limitations, and preferences on quality, cost, and duration. It is a trade-off analysis expert and what makes this agent, and BIG, able to meet real-time deadlines and to perform differently in different circumstances, e.g., flexible degradation in the face of limited time. BIG's trade-off behaviors are documented in [17, 9].

Execution Control Subsystem The output of the Design-to-Criteria (DTC) scheduler is a fully ordered sequence of actions for the agent to carry out. In contrast to BIG, where certain actions are performed in parallel, the parallelism feature of DTC is not currently used in our agent (to simplify the construction of the execution subsystem). The execution subsystem combines the schedule with the action dependency specification and the actions (contained in the action library) and executes the schedule.

Internet Site Wrappers Actions performed during execution basically reduce to invoking a specific web site wrapper (associated with the action in question) and passing along appropriate parameters. For example, one might invoke the weather.com wrapper with a specific zip code to obtain the five day forecast for a given area. The wrappers handle the http get requests, grab the html output, and process the output to obtain structured data from the

unstructured text. This data is inserted into trip objects, which contain fragments of itineraries or partial itineraries.

Decision Maker The trip objects produced by the wrappers/execution subsystem are passed to the decision maker, described in greater detail in Section 3.4, which uses the client preferences to construct and select itineraries for the client. In future versions of this work, the decision maker will be invoked during execution and will possibly provide feedback to the user who can then refine his or her preferences and modulate the search as it progresses. This control concept was also targeted for BIG though never implemented.

Of these components the research grade technologies are the Design-to-Criteria scheduler, the query processor, the capability assessor, and the decision maker. The Design-to-Criteria scheduler is a mature artifact that has been in service, but evolving, since 1995. The other components are notable in that they differ significantly from those used in BIG or they are completely original. We thus discuss them in greater detail in the following subsections.

3.2 Query Expansion

The values of the keywords part of the user's query will be expanded based on information encoded in the semantically organized lexicon WordNet [5] as well as in our own theme-based lexicons. For example of the use of WordNet, if a user includes "mountaineering" as a keyword in the recreation category, then a closely synonymous activity – one of mountaineering's "senses" such as

”climbing” could be added to the query, with lower priority in the search than primary keywords.

The use of these lexicons enables a variety of arbitrary distance calculations to determine a set of words related to the keywords. We are examining a calculation of closeness of two terms based on a combination of quality and similarity distances through the relations provided by the lexicons. Priority grading *x has priority over y* and similarity *x is more similar to z than y* functions, in addition to some hand-crafted, domain-dependent weights based on trip related ”scenes” guide the selection of different sets of keyword sets used to query. For an example of the latter information a beach scene might invoke the set {”towel”, ”ball”, ”coast”, ”fish”, ”trail”}.

The expanded set of keywords may then be used to expand the information gathering phase of the agents activities. It will initially use the following strategies to focus its search at these choice points:

1. Closeness of expanded set of keywords to the original set based on similarity and quality.
2. Closeness of match between expected information gathered and user’s query, i.e., each information source has a type associated with it. The name of that type in the semantic lexicon could provide the basis for this measure.

3.3 Capability Assessor

The problem facing a situated information gathering agent is that of deciding what actions to perform that adequately satisfy a user’s query in a timely fashion. There are different approaches to this problem. However, today it is widely held that a planning component is essential [27] to provide the flexibility and customizability required for complex and variable user queries. There are a variety of ways that this component has manifested itself, i.e., as an extension to UCPOP [12] or as a mix of components in BIG [16]. The latter approach is of particular interest to us since we too seek to leverage the sophisticated reasoning provided by a DTC scheduling component.

DTC provides a way of selecting and scheduling statistically characterized sets of agent actions organized using the TÆMS language. Visually, TÆMS is a tree that describes the hierarchical decomposition of tasks (and goals) into subtasks and finally into the primitive actions that can be used to carry them out. TÆMS task structures serve as the input to DTC – it is how DTC understands the agent’s problem solving options. Our view of the role of TÆMS is evolving and we currently like to view a single-agent TÆMS task structure as an abstract characterization of an agent’s capabilities at a particular instant in time and within a particular (larger) problem solving context. The task structure specifies what actions

the agent may carry out and how these relate by defining the agent’s problem solving structure, e.g., which tasks affect others, precedence constraints, how quality propagates through the graph, etc. However, TÆMS task structures are deliberately abstract – though TÆMS actions are characterized via discrete probability distributions in terms of quality, cost, and duration, the internal details of the actions are omitted. Actions in TÆMS are essentially black boxes. This abstraction is sufficient to perform scheduling and analysis of TÆMS task structures, or even coordination between multiple agents, but, more detail is necessary to actually carry out the actions in the task structure.

Though rich in some respects, a static TÆMS task structure is inadequate to wholly describe the entire range of problem solving options of a complex agent in all circumstances and for all time. While one could actually fully enumerate all of an agent’s options for all points in time via TÆMS, the structure would be enormous as TÆMS does not fully support concepts like loops and the representation is stateless. TÆMS is perfectly well suited to be emitted by a complex planner or problem solver, but, in the general case, it is insufficient in and of itself to replace a planner or problem solver. Some of the key problems are:

1. *No support for abstract action types* – although the possible actions represented in a TÆMS task structure are abstractions by virtue of their statistical characterizations, they are still instantiations of more abstract action types which may have a range of situation-dependent statistical characterizations.
2. *Situation not explicitly recognized* – even if one was to enumerate extensive sets of different functionally grouped actions with situation-dependent characterizations, there is no explicit support in TÆMS to choose the proper set dependent on the situation.
3. *Fixed structure, variable functionality* – there are complex, situation-dependent relationships between functional decompositions of agent actions in TÆMS that TÆMS cannot itself represent, i.e., it has no metalanguage support.

An example from the aforementioned application domain is used to illustrate these limitations. A TripBot user might eventually specify as part of a larger query that they are not particularly interested in plane travel, would rather camp on a beach than stay in a hotel, is interested in visiting a wide variety of high-quality restaurants, and would like TripBot to take at least an hour trying to find a good trip that meets his or her requirements.

It is possible to provide a TÆMS task structure template that would address meeting the broad objectives of

the above user's query. That is, part of its decomposition could contain tasks relevant to gathering information about what means of travel are available to his or her destination in addition to what accommodations are available in that location, including, possibly, hotels and camping. Other tasks could address, broadly, finding restaurants in the area. There are some problems with the template approach. We will examine two: focus and flexibility.

The problem of focus when actions from a task structure is one of where to focus to maximize the quality of the agents actions. Let's consider the naive case first. The agent could be given an unmediated generic TÆMS task structure for gathering trip information. In this case, the agent's initial schedule would undoubtedly contain actions that could add little to the goal of satisfying a particular user's query parameters. That is, using a generic structure, more information about hotels than camping might be gathered.

One step up from this scheme would be one in which a generic TÆMS task structure is mediated by a model which might adjust the quality distributions of various actions under a prescribed functional decomposition. Such a mediation system might set the quality of irrelevant actions to zero, thus effectively pruning them from the space of considered actions. The problem here is that such a scheme, if done without potentially complex computations about the interactions between the "pruned" actions and other actions that are part of the initial structure may dramatically change the range of schedules available from the task structure in undesirable ways. This problem can arise because of the arbitrarily complicated semantic mapping between the actual agent capabilities and environments and their TÆMS abstractions.

The problem with both schemes is that it still provides no way of expanding the given range of actions to include more options related to the user's preferred outcome. In this same vein, the parameterized template provides little support for opportunism or handling of failure conditions, i.e., support for assessing the agent's capabilities in terms of a TÆMS task structure in a context-sensitive manner. Moreover, the route from a complex query involving preferences like those given above to a TÆMS task structure is a potentially complicated one. That is, constructing an adequate characterization of a goal from a user's stated query parameters and then using that goal as the basis for search through a space of agent capability descriptions is, in general, a task that requires a more sophisticated approach than those described so far.

We have decided to approach the problem of characterizing an agents capabilities at a given instance and within a given goal context as a planning problem. However the formulation of the problem and its solution differ in some important ways from other planning problems. One difference is that the resulting structure is not an or-

dered sequence of actions, but rather a family of hierarchically and functionally organized, statistically characterized components of such sequences – TÆMS task structures. Another difference is that only the semantics of task and method interactions must be modeled, potential interaction dependencies and resource contentions between action sequences that would normally have to be resolved by a planner are resolved by the DTC scheduler.

The CA task structure generation algorithm resembles that of an ordered HTN planner [18] in that each path of the generated task structure is decomposed according to specified decomposition operators until primitive actions are reached, at which point the decomposition operations halt on that branch; a subset of applicable actions is then included at that point in the task structure. The task decomposition algorithm requires a high-level task as its input. The initial world state is empty unless otherwise specified in the input parameters. It then uses task operators and methods provided in a domain information file to decompose the initial task into a set of task substructures describing alternatives for satisfying the top-level-task. A simplified example follows.

```
(spec_task (name Create_Trips))
```

The CA first looks for a method with the name *Create_Trips*. Failing to find one, it looks for an operator with that name. It might have the following operators:

```
(spec_operator
  (name Create_Trips)
  (decomposition
    Get_Query_Info
    +Gather_Trip_Info
    Make_Decision))
```

```
(spec_operator
  (name Gather_Trip_Info)
  (decomposition
    +Get_Flights
    +Get_Hotels
    +Get_Rental_Cars))
```

Here, the CA would choose the *Create_Trips* task decomposition. This would replace the *Create_Trips* task at the head of the list of tasks to expand into at least three new tasks: *Get_Query_Info*, *Gather_Trip_Info*, and *Make_Decision*. The '+' sign in front of a task name in the decomposition field of an operator is to indicate that the decomposition may contain one or more of that task.

To complete our simple example, the CA would then choose available methods which complete the above tasks. For example, it might choose the following method to add to the *Get_Flights* task.

```

(spec_method_info
  (name Query_AOW_Flights)
  (family Get_Flights)
  (constructor_params
    (:internal Executor)
    (:internal Duration)
    City
    City
    Date
    Date
    AirClass
    Airline)
  (results AirFlight)
  (outcomes
    (outcome_1
      (density 1.)
      (quality_distribution
        0 .20 10 .80)
      (duration_distribution 40 1.0)
      (cost_distribution 0 1.0)))
  (implementations
    (implementation_1
      (platform Java)
      (class
        wrappers.AOWFlightWrapper)
      (method AOWFlightWrapper))))

```

The above `QueryAOWFlights` method would be added under the `Get_Flights` task in the generated TÆMS task structure by creating a unique symbol for the method label in the structure, appending its statistical distributions, and adding the dependencies between it and other methods or tasks, if there are any. These interrelationships create a resource or information flow dependency graph that contains information for every possible schedule's method/parameter dependencies, and is used by the execution subsystem to track resources generated or required by the instantiated actions of a schedule in a resource table.

We are working on a version of the CA that will handle disjunctive requirements and complex return types and a more refined system of quantification. There are also some additional complexities endemic to task decomposition within the TÆMS formalism because of the fairly extensive variety of relations supported between tasks, subtasks, and actions and we are incrementally incorporating them as we find need for them in the application domain. Since speed is a concern, we are trying to effectively balance expressiveness and efficiency in our approach. Initially, we have focused on producing a fast, expressive planner, i.e., knowledge-based, forward-searching, and total-ordered, but we are looking more seriously now at expanding its cross-domain capabilities and mitigating the amount of knowledge engineering required to model decompositions.

3.4 The Decision Making Module

One of the key components in a personal agent, such as TripBot, is the decision maker. Typically it is the decision maker that reasons about gathered information and the artifacts produced by combining the information. In the BIG information gathering agent, fusion is the job of the blackboard problem solver whereas the decision maker's function is primarily evaluation. In the TripBot, the decision maker also performs fusion because the fusion process is intertwined with user preferences evaluation, i.e., TripBot must use user preferences to control combinatorics in order to produce a final set of objects from which to return one (or a ranked set) to the user.

The decision making research has several different facets: 1) the specification of user preferences which we call *constraint selection*, 2) the use of preferences in problem solving, which we call *evaluation*, and 3) controlling combinatorics to produce a set of itineraries appropriate for the client.

3.4.1 Constraint Selection

In the TripBot implementation, the user preferences can be set as certain constraints over a set of individual and overall factors. Generally, these constraints can be defined as cost, quality, and duration. Often these constraints are interrelated or even conflict, therefore, the technique of "Constraint Satisfaction Problems" (CSP) was applied in the decision-making module to get more optimal trip alternatives.

There are two types of constraints that users can specify when they initialize a request to the TripBot, these constraints are so called "hard constraints" and "soft constraints" [11]. Hard constraint is a threshold that indicates the upper bound or lower bound of a specific factor, such as "I am not willing to spend more than \$700 dollars for this trip". Soft constraint can be used to express user's criteria as a scale of preferences using weights, such as "the importance of the cost is 20% and the importance of the hotel quality is 80%", which can be interpreted as "but the hotel quality is more important than the cost".

Hard constraints are being used in most of the online travel service sites, where users can only specify the "hard" thresholds such as departure date, cost limit, specific airline name, and hotel chain etc. before the search. The problem of hard constraint is that it may cut out some potentially more optimal candidates. For example, if there are two trip plan A and B, both of them offer identical services except that the plan A includes a 5-star hotel stay with total cost of \$705, and the plan B includes a 1-star hotel stay with total cost of \$690. Apparently the plan A looks better than the plan B. But if the user only sets a hard constraint saying that the total cost can not be over \$700, then the plan A will be pruned and the plan B

will be selected. Therefore, making a decision only based on hard constraints is not very flexible.

Soft constraint is more flexible since it can express user's relative preference without specifying the details. It allows sub-optimal solutions to remain in the search space in order to get more relatively optimal result. But the problem of soft constraint is that it lacks some kind of standard while performing comparison. In another words, sometimes it is hard to know how good is good enough or how bad is too bad. Back to the above example, if this time plan A includes a 5-star hotel with total cost of \$3000, and plan B includes a 1-star hotel with total cost of \$690. Usually we draw a conclusion that the plan A is too expensive to stay. But if the user only sets a soft constraint saying that the hotel quality is much more important than the cost, then the plan A probably will be selected despite the fact that the user may not be able to afford it.

Therefore, in the decision-making module of TripBot, we associate these two types of constraints together in order to get more optimal and flexible solutions. Figure 4 to Figure 6 shows the input interface of the TripBot, where the agent asks a user to specify some criteria of the trip. Among them, destination, departure time, preferred airline, and total cost limitation etc. are hard constraints, and the importance of hotel quality, flight quality, rental-car quality and the importance of total expense limit etc. are soft constraints. Rather than arbitrarily separates out acceptable itineraries, decision module computes a ranking number for each itinerary that reflects the degree for which both types of constraints are satisfied. By this way, we can keep a larger and more reasonable itinerary search space to let the decision making module work on, and hope it can generate more optimal solutions. Back to the previous example again, this time the agent gets three possible trip candidates. Plan A, B, and C have identical services except that the plan A includes a 5-star hotel with total cost \$3000, the plan B includes a 1-star hotel with total cost \$690, and the plan C includes a 4-star hotel with total cost \$750. If we apply both hard and soft constraints such as "the cost limit for the trip is \$700" and "the hotel quality is much more important than the cost" during the decision making, then a TripBot probably will assign a higher ranking number to the plan C if the agent has been properly knowledge engineered.

3.4.2 Evaluation Algorithm

In order to recommend the best itinerary to the user, the itineraries must be evaluated and compared to each other. One complete itinerary consists of many different services such as air transportation, hotel stay and rental car etc. The problem during the evaluation is that these services are often with competing and even conflicting con-

straints. When re-composing an itinerary from these services, a choice may be optimal for some of the criteria but the same choice may become the worst solution for others. The decision-making module attempts to solve this situation by applying the utility theory to generate a numerical score to rank the solutions. The idea of utility theory is to use "utility" to quantify the "quality of being useful" [19]. In the decision-making module, utility functions are used to map the degrees of constraint satisfaction to real numbers. A `getUtility()` function is implemented for each travel object and is called during the decision making process. What the utility function does is to compute the utility of a specific service based on a set of user preferences passed from the user-input interface. Then the overall utility of an itinerary can be calculated as the weighted sum of the utilities of all constrained services. Also in the decision-making module, hard constraints are formulized to compute the absolute utilities and soft constraints are formulized as the weights for certain utilities. The idea can be illustrated using the following equation: $U_{itit} = \sum p_i U(S_i)$, Where U_{itit} is the overall utility of a specific itinerary, S represents a particular service, $U(S_i)$ returns the utility of a specific service, and p_i is the weight of a specific utility and defines how important it is relative to the others.

To simplify the model specification, it is assumed that a set of services is mutually utility-independent. Therefore, the preferences over the individual services do not depend on the level at which the other services are achieved. Table 1 illustrates some of the constraints that are currently being reasoned during the decision making process:

Although most preferences are explicitly indicated by the user in the input form, the agent still keeps some default preferences. For instance, if there are two itineraries with similar quality, by default the agent will favor the cheaper one despite that user does not set any cost preference. The agent also assumes that shorter flying time, less number of stops and higher quality of hotel service etc. are preferred by the user.

3.4.3 Constraint Satisfaction Search Algorithm

By applying the utility functions we can have every possible itinerary been evaluated. Then the itineraries with highest ranks can be picked out as trip recommendations. However, there is something wrong with this scenario. Let's consider a simple trip planning which only consists of 40 possible air flights, 40 possible hotel stays, and 40 possible rental cars. There are total $40 \times 40 \times 40 = 64000$ different itineraries. In order to get the most optimal solution, we have to compose all these itineraries and then evaluate them. It might be fine to compute 64000 alternatives in a modern machine, but how about if we have to

	Flight	Hotel Stay	Rental Car	Total Expense
Quality	Departure time Flying time # of stops Airlines	Hotel location Hotel chain Hotel service	Rental company Car type	
Cost	Flight price	Hotel price	Rental price	Total cost
Weights	Importance of the flight quality	Importance of the hotel quality	Importance of the rental car quality	Importance of the cost limit

Table 1: The constraints being evaluated in the decision-making module

reason about 10 different services and each service has 100 different instances? It is just simply too much to compute.

Basically, this is a Constraint Satisfaction Problem (CSP). Therefore, our approach is to make the decision-making module acts as a constraint solver, which applies proper searching algorithms to effectively reduce the search space without losing too much optimization [22]. The specific searching technique currently used in the decision module is Forward Checking. The algorithm can be simply illustrated as following: First, each service object will be partially evaluated and be pre-sorted based on its service utility in each service category. The objective of the pre-sorting is to prepare the search space for the CSP solver. Then the forward checking algorithm is applied to effectively reduce the size of CSP by looking ahead and pruning all of those itineraries that may violate the overall utility threshold. One simple trial run shows that this algorithm successfully reduced a 10x10x10 search space to 126 instantiated itineraries. Among them, ten itineraries with highest utility scores were selected and presented to the user.

3.5 Extended Example

To demonstrate the operation of this system, we provide an example of using the system to plan a family vacation. When the destination of the trip is unconstrained, possibly the most entertaining and useful function of the TripBot is to suggest destinations that fit some input criteria.

For example, the system begins with the user stating that he wishes to travel from Boston to the Southeast with his family (2 adults and 1 child) on August 10. They want to spend a week on the vacation and also want to limit the total expense (transportation, accommodation etc.) to \$3,000. They wish to visit somewhere with a beach and an amusement park nearby. Their daughter enjoys movies, and all of them would like to attend at least one concert. They enjoy Italian and Chinese food. They also specify other preferences like the airline, hotel chain, rental car type and rental company. Several soft constraints can also be specified such as the importance of

flight quality, hotel quality, rental car quality, and the importance of the total cost. The agent then converts these preferences into a user model, adding default preferences for unspecified attributes. Default preferences are common to almost all users. For example, most users prefer higher quality services. But between two services with similar quality, most users want the cheaper one. Most users prefer non-stop flights rather than changing planes several times during the trip etc.

Though user preferences are used in the decision-making component in the later stages of the agent’s process, they are also used early in the process to target the agent’s activities. In the current implementation, these preferences are passed as input to the agent’s planning component, which decides where to find information resources, when to use them, in what order, and so forth. After the agent collects all the necessary information, it will invoke its decision making component to construct itineraries based on user preferences so that only “good” itineraries are constructed and evaluated, which is done by applying a constraint satisfaction search algorithm. By focusing on (potentially) “good” itineraries, the agent can avoid generating too many possible combinations, since in most cases producing and evaluating all itineraries is not feasible due to time and computational limitations.

In the above example, first the agent will generate several city candidates associated with the trip type and region. Then for each destination city, the agent will collect related information and construct and evaluate possible itineraries. In the current implementation, four variables are considered in the decision-making. They are flight quality, hotel quality, rental car quality and total cost quality. The algorithm proceeds by assigning each flight, hotel and car variable with a value, which is calculated by utility functions. For example, a specific flight variable will be evaluated based on its departure time, total flying time, the number of stops, the name of the airline, and its cost. Therefore, a straightforward depth-first search algorithm can be applied to construct and evaluate possible itineraries composed from these variables. The utility of each itinerary is a weighted sum of flight utility, hotel quality, car quality, and the total cost quality. To reduce the search space, an overall itinerary utility thresh-

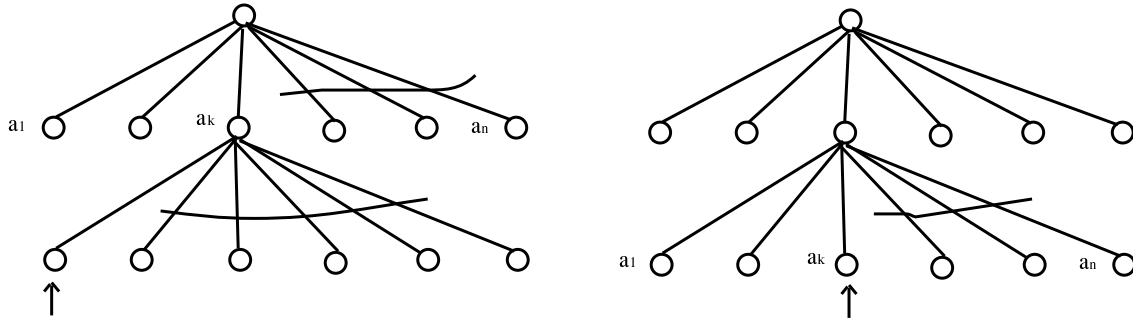


Figure 2: Pruning During Itinerary Construction

old can be set to distinguish those acceptable itineraries, and pre-evaluation and sorting of all flights, hotels, and cars can be conducted based on their utility values. Note that some pruning can take place in the search tree. For example, suppose the values of variable A are ordered as $a_1 \dots a_n$. If assignment $A = a_k$ and all of its children do not satisfy the itinerary utility threshold, then we do not continue to search $A = a_i$ where $(k < i < n)$, as illustrated in Figure 2. Finally, for each destination city, three top-ranked itineraries will be displayed along with their utility values and other additional information, such as local weather, local movie theaters, concerts in the area, and local restaurants with user's favorite cuisine. Satisfied, the user ends the session.

It is possible that a potentially qualified itinerary will never be produced because the candidate components are pruned during the intermediate phases of the process. This is because the total trip cost depends on the costs of the trip components. However, when ordering flight, hotel, and car variables, the total cost is an unknown quantity, i.e., these activities are treated as being independent to avoid having to generate all possible itineraries in order to compute the total cost. The end result is that we may prune out some potentially qualified itinerary with variable A near a_k . This is an example of making a set of local optimizations that are not guaranteed to combine to a globally optimal solution.

While we have not yet fully addressed this issue, some improvements have been proposed. One possible work around is to adjust the itinerary utility threshold. A lower threshold means more itineraries will be constructed and evaluated, therefore decreasing the odds that an optimal itinerary will be pruned away. Another option is to set the threshold dynamically based on the size of the space involved.

Returning to the example, at the end of the process, for each destination city, three top-ranked itineraries will be displayed along with two extreme cases, the cheapest itinerary and the highest quality. Though there are many other candidate mixes or heuristics that could be used to provide a range of choices for the client.

3.6 Wrappers

We use the expression *wrapper* to denote the code that conceptually wraps Internet sites for our agent. We create wrappers as a way to abstract away the irregularities of the Internet to a common interface. Each wrapper takes in a set of parameters and returns a vector of data structures compiled from an Internet site. When the execution subsystem reads in a requested wrapper call from the schedule, as shown in Figure 3, it first looks up the action associated with the scheduled TÆMS method in the action library. The action is then referenced into the dependency graph and the action dependencies (parameters and return objects) are returned from the resource database. The wrapper is then instantiated with the required parameters and given a maximum execution time. When either the wrapper has finished gathering information, or the time limit has expired, the internal data set of the wrapper is returned to the execution subsystem and entered into the resource database.

In this implementation we use static website wrappers that use a simple form of pattern matching to parse the result of queries. For example, a wrapper that finds concerts for a given city might take as a parameter a date and the name of a city. The wrapper would then perform a query on a concert venue search engine such as Pollstar.com and parse the returned information for concert objects. This concert wrapper has an internal representation of what a concert object looks like in an HTML page, and searches the document returned from the query for each instance of the representation. The internal representation is a set of string delimiters, one for each member in the object.

Figure 4 shows how the Pollstar concert wrapper identifies concert objects in a page returned by Pollstar.com. The wrapper identifies vertically spaced entries by location delimiting strings around concert fields. First the wrapper finds a date field by finding its delimiting strings in HTML, then the artist field, and finally the venue field. After it has identified these fields it can populate a concert object and add it to its internal data set. After the wrapper can no longer find any more date fields (or when its timer

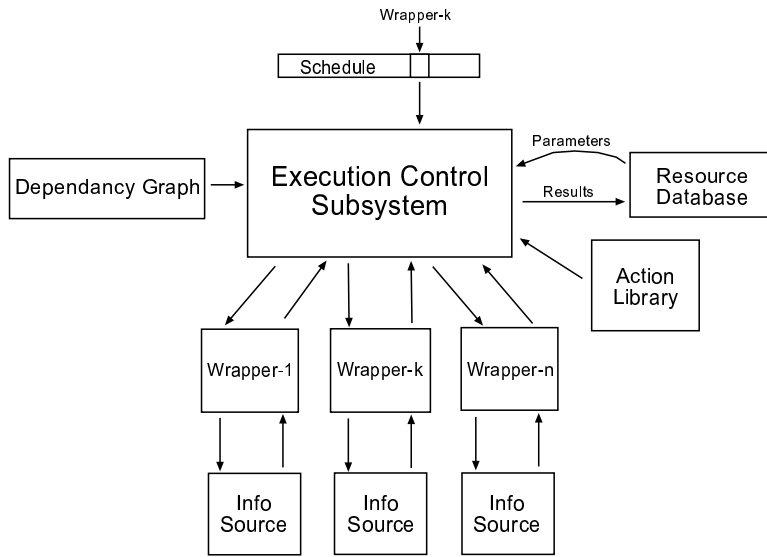


Figure 3: Wrapper Invocation

TOUR INFORMATION
Bangor, ME
 All Dates Selected. [Limit Date Range](#)
 Found 4 Items [Print](#)
[Include surrounding cities](#)

Concert Object		
Date	Artist	Venue
Sat 05/19/01	Lynyrd Skynyrd	Bangor Auditorium & Civic Center
Sat 05/19/01	Ted Nugent	Bangor Auditorium & Civic Center
Sun 07/29/01	Tony Trischka	Bangor State Fair
Fri 08/03/01	Andy Griggs	Bangor State Fairgrounds

Date Field Artist Field Venue Field

```

addToDataSet( new Concert("Lynyrd Skynyrd", "Bangor Auditorium & Civic Center", "05/19/01") );
  
```

Figure 4: Wrapping for Concert Information

runs out) the wrapper returns the objects populated from the query to the execution subsystem. A future expansion might be to create a system that can dynamically create wrappers, or learn how to wrap web sites autonomously, as in [8].

4 Implementation Overview

The TripBot agent is currently running on an 800mhz Pentium III, with 128MB Ram under Red Hat linux 7.0. When it is put online (at mas.umcs.maine.edu) it will migrate to a machine with dual 1ghz processors under linux. The larger machine has more more RAM and a SCSI drive subsystem, however, it is worth noting that the TripBot will run well on machines with much less processing power than the 800mhz Pentiums being used for development. Most of the TripBot is implemented in Java, though the Design-to-Criteria scheduler is written in C++, the Capability Assessor uses Allegro Common Lisp(ACL) 6.0, and part of the user interface uses Perl.

Screen snapshots of the GUI are shown in Figures 5, 6, 7. The current implementation status is that each of the components has been tested in the small and most of the components are currently working together properly. We can execute the front part of the agent problem solving process and the latter stages independently – the decision maker and execution subsystem components are currently being integrated and the integration does not involve any research issues. In the future, we would like to distribute the key components used in the TripBot and to create estimates for the resources required to customize them for other information gathering applications.

5 Conclusion and Future Directions

We have presented a new view on real-time information gathering agent construction that differs from the approach used in the BIG information gathering agent. Essentially, we have eliminated and replaced the components that were necessary to address arbitrary search, discovery, and processing online because they were also difficult to adapt to new domains and less predictable than other, less ambitious, technologies. We also replaced the opportunistic internal control and the pseudo-hand-generated process prototypes used in BIG to produce TÆMS task structures with the capabilities assessor / planning component that generates TÆMS task structures from first principles.

The modified architecture is being deployed on crisis management, travel planning, and music-centered digital libraries projects. The advantage of using agent technology in these applications is that it enables the software to meet soft real-time deadlines, respond to the dynamics of

the Internet environment, reason about client preference, and plan to achieve the objectives.

Implementationally, this work is in its formative stages. However, we are building on a strong research foundation set by the BIG agent and the Design-to-Criteria scheduler is a mature technology. We believe the architecture described here will address the requirements of the above applications without significant modifications.

On the research front, aside from the overall architecture and view of the problem space, the decision module uses a new CSP approach to online trip planning where user criteria and preferences are explicitly modeled as combination of hard and soft constraints. Preferences and evaluation are also used in the decision module to control combinatorics. The end result is that the TripBot allows a user to examine a large set of possible solutions in a reasonable amount of time. In the future, we hope to show that TripBot outperforms any existing tools on this front and is faster than human driven online travel planning.

There are plenty of open questions and areas for improvement in this research. In the decision-making process, other heuristic search algorithms may be explored in the future. Additionally, the TripBot currently uses only a limited number of travel related information sources. In the future, more travel sites should be added to the TripBot's list of resources to enhance flexibility, improve error recovery, and to potentially improve accuracy. The agent could also use a more flexible integration of control, decision-making, and user preference specification – ideally the user should be able to see what the agent does, while it does it, and provide guidance or feedback online. There is also a great deal of room to improve personalization of the TripBot's services, e.g., storing user preferences and recording default values. There is also room to explore additional value-added services that the TripBot agent could provide. For example, destination suggestion, travel route planning, and the integration of driving directions and other classes of information into the basic agent.

6 Acknowledgements

We would like to acknowledge the efforts of the UMASS BIG agent team members who did not participate directly in this work but from whose work some of these ideas are derived. The BIG agent was constructed by Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley XQ. Zhang. Aspects of the work presented here are also being deployed in a related digital library project and the collaborators on that effort include Roy and Elise Turner, also of MaineSAIL, and Tom Wheeler. We would also like to thank other researchers who have continued to help evolve TÆMS and TÆMS based con-

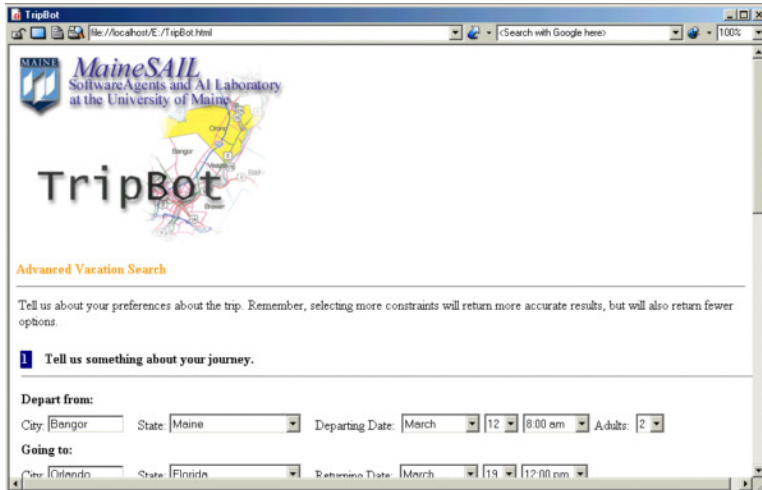


Figure 5: The TripBot Agent GUI

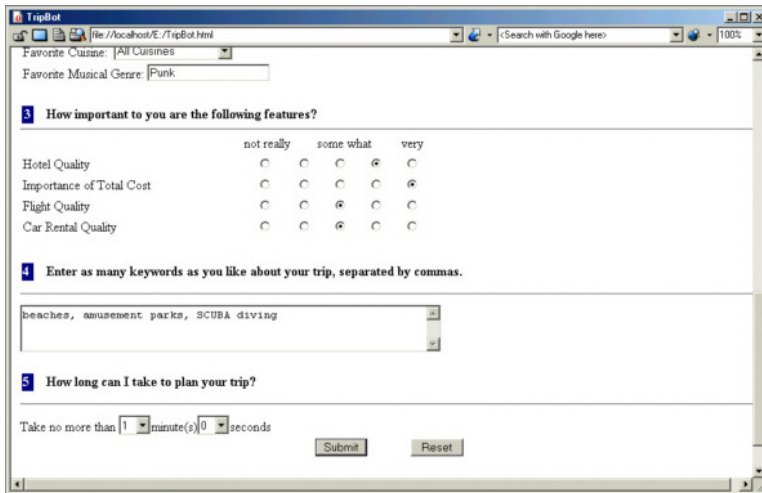


Figure 6: The TripBot Agent GUI, Part 2

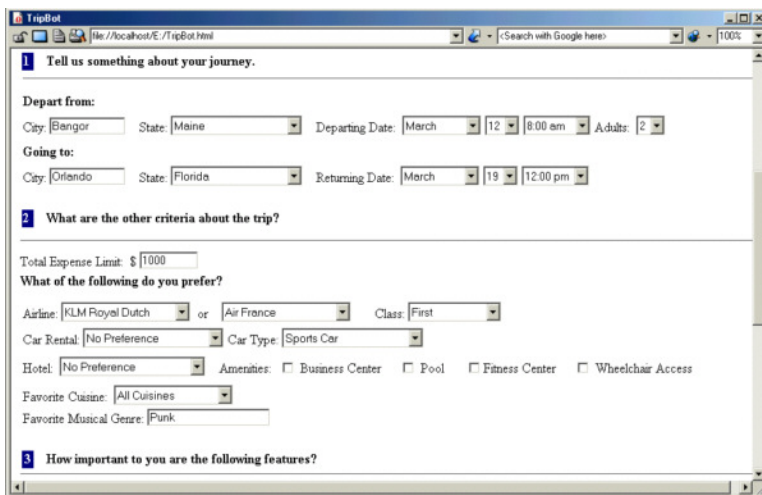


Figure 7: The TripBot Agent GUI, Part 3

trol of software agents, including Victor Lesser, Keith Decker, Bryan Horling, Regis Vincent, Ping Xuan, Shelley XQ. Zhang, Anita Raja, and Roger Mailler.

References

- [1] Daniel Bilus and Michael J. Pazzani. A Personal new Agent that Talks, Learns, and Explains. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, 1999.
- [2] J. P. Callan, W. Bruce Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of the 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.
- [3] Norman Carver and Victor Lesser. A new framework for sensor interpretation: Planning to resolve sources of uncertainty. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 724–731, August 1991.
- [4] Norman Carver and Victor Lesser. A planner for the control of problem solving systems. *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Planning, Scheduling and Control*, (6):1519–1536, 1993.
- [5] F. Christiana. *WORDNET: an electronic lexical database and some of its applications*. Cambridge, MA: MIT Press, 1999.
- [6] K. Decker, A. Pannu, K. Sycara, and M. Williamson. Designing behaviors for information agents. In *Proceedings of the 1st Intl. Conf. on Autonomous Agents*, pages 404–413, Marina del Rey, February 1997.
- [7] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on “Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior”.
- [8] Robert Doorenbos, Oren Etzioni, and Daniel Weld. A scalable comparison-shopping agent for the world-wide-web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, Marina del Rey, California, February 1997.
- [9] Victor Lesser et al. Sophisticated Information Gathering in a Marketplace of Information Providers. *IEEE Internet Computing*, 4(2):49–58, Mar/Apr 2000.
- [10] Oren Etzioni. Moving up the information food chain: Employing softbots on the world wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1322–1326, Portland, OR, August 1996.
- [11] R. L. Keeney and H. Faiffa. *Decision Making with Multiple Objectives: Preferences and Value Trade-offs*. Cambridge University Press, Cambridge, UK, 1993.
- [12] Craig A. Knoblock. Building a planner for information gathering: A report from the trenches. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 134–141. AAAI Press, 1996.
- [13] Bruce Krulwich. The BargainFinder Agent: Comparison price shopping on the Internet. In Joseph Williams, editor, *Bots and Other Internet Beasties*. SAMS.NET, 1996. <http://bf.cstar.ac.com/bf/>.
- [14] Leah Larkey and W. Bruce Croft. Combining classifiers in text categorization. In *Proceedings of the 19th International Conference on Research and Development in Information Retrieval (SIGIR '96)*, pages 289–297, Zurich, Switzerland, 1996.
- [15] Victor Lesser, Michael Atighetchi, Bryan Horling, Brett Benyo, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. A Multi-Agent System for Intelligent Environment Control. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, 1999.
- [16] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley XQ. Zhang. BIG: A resource-bounded information gathering agent. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, July 1998. See also UMass CS Technical Reports 98-03 and 97-34.
- [17] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley XQ. Zhang. BIG: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence*, 118(1-2):197–244, May 2000. Elsevier Science Publishing.
- [18] D. Nau, Y. Cao, A. Lotem, and H. Muoz-Avila. Shop: Simple hierarchical ordered planner. In *IJCAI-99*, pages 968–973, 1999.
- [19] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

- [20] Richard B. Segal and Jeffrey O. Kepar. MailCat: An Intelligent Assistant for Organizing E-Mail. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, 1999.
- [21] S. Soderland, D. Fisher, J Aseltine, and W.G. Lehnert. Crystal: Inducing a conceptual dictionary. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1314–1321, 1995.
- [22] M. Torrens, R. Weigel, and B. Faltings. Java constraint library: Brining constraint technology on the internet using the java language. In *Proceedings of the Workshop on Constraint Reasoning on the Internet*, 1997.
- [23] et al Victor Lesser, Bryan Horling. The TÆMS whitepaper / evolving specification. <http://mas.cs.umass.edu/research/taems/white>.
- [24] Thomas Wagner, Alan Garvey, and Victor Lesser. Complex Goal Criteria and Its Application in Design-to-Criteria Scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 294–301, July 1997. Also available as UMASS CS TR-1997-10.
- [25] Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19(1-2):91–118, 1998. A version also available as UMASS CS TR-97-59.
- [26] Thomas Wagner and Victor Lesser. Design-to-Criteria Scheduling: Real-Time Agent Control. In Thomas Wagner and Omer Rana, editors, *To appear in Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, LNCS. Springer-Verlag, 2001.
- [27] D. Weld. Planning-based control of software agents. In *Proc. 3rd Intl. Conf. AI Planning Systems (AIPS)*, 1996.