Getting from Here to There: Interactive Planning and Agent Execution for Optimizing Travel

José Luis Ambite, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292, USA
{ambite,barish,knoblock,mariam,jeanoh}@isi.edu

Steven Minton
Fetch Technologies
4676 Admiralty Way,
Marina del Rey, CA 90292, USA
steve.minton@fetch.com

Abstract

Planning and monitoring a trip is a common but complicated human activity. Creating an itinerary is nontrivial because it requires coordination with existing schedules and making a variety of interdependent choices. Once planned, there are many possible events that can affect the plan, such as schedule changes or flight cancellations, and checking for these possible events requires time and effort. In this paper, we describe how Heracles and Theseus, two information gathering and monitoring tools that we built, can be used to simplify this process. Heracles is a hierarchical constraint planner that aids in interactive itinerary development by showing how a particular choice (e.g., destination airport) affects other choices (e.g., possible modes of transportation, available airlines, etc.). Heracles builds on an information agent platform, called Theseus, that provides the technology for efficiently executing agents for information gathering and monitoring tasks. In this paper we present the technologies underlying these systems and describe how they are applied to build a state-of-the-art travel system.

Introduction

The standard approach to planning business trips is to select the flights, reserve a hotel, and possibly a car at the destination. Choices of which airports to fly into and out of, whether to park at the airport or take a taxi, and whether to rent a car at the destination are often ad hoc choices based on past experience. These choices are frequently suboptimal, but the time and effort required to make more informed choices usually outweighs the cost. Similarly, once a trip has been planned it is usually ignored until a few hours before the first flight. A traveler might check on the status of the flights or use one of the services that automatically notify a traveler of flight status information, but otherwise a traveler just copes with problems that arise as they arise. Beyond flight delays and cancellations there a variety of possible events that occur in the real world that one would ideally like to anticipate, but again the cost and effort required to monitor for these events is not usually deemed to be worth the trouble. Schedules can change, prices may go down after purchasing a ticket, flight delays can result in missed connections, and hotel rooms and rental cars are given away because travelers arrive late.

To address these issues we have developed an integrated travel planning and monitoring system. The *Travel Assistant* provides an interactive approach to making travel plans where all of the information required to make informed choices is available to the user. For example, when deciding whether to park at the airport or take a taxi, the system compares the cost of parking and the cost of a taxi given other selections, such as the airport, the specific parking lot, and the starting location of the traveler. Likewise, when the user is deciding which airport to fly into, the system provides not only the cost of the flights, but also determines the impact on the cost of the ground transportation at the destination. Once a trip is planned, the monitoring tasks are addressed by a set of information agents that can attend to details for which it would be impractical for a human assistant to monitor. For example, beyond simply notifying a traveler of flight delays, an agent will also send faxes to the hotel and car rental agencies to notify them of the delay and ensure that the room and car will be available. Likewise, when a traveler arrives in a city for a connecting flight, an agent notifies the traveler if there are any earlier connecting flights and provides both arrival and departure gates.

These innovations in travel planning and monitoring are made possible by two underlying AI technologies. The first is the Heracles interactive constraint-based planner (Knoblock *et al.* 2001), which captures the interrelationships of the data and user choices using a set of constraint rules. Using Heracles we can easily define a system for interactively planning a trip. The second is the Theseus information agent platform (Barish *et al.* 2000), which facilitates the rapid creation of information gathering and monitoring agents. These agents provide data to the Heracles planner and keep track of information changes relevant to the travel plans. Based on these technologies, we have developed a complete end-to-end travel planning and monitoring system that is in use today.

The remainder of this paper describes the travel application and underlying technology in more detail. The next section describes by example the trip planning process as well as the monitoring agents that ensure that the trip is executed smoothly. Then, we present the constraint-based planning technology that supports the trip planning. Next, we describe the information agent technology, which provides the data for the planner and the agents for monitoring the trip.

Finally, we compare with related work, and discuss our contributions and future plans.

Planning and Monitoring a Trip

In this section we describe by example the functionality and interface of our *Travel Assistant*, showing both its capabilities for interactive planning and for monitoring the travel plans.

Interactive Travel Planning

Our *Travel Assistant* starts the travel planning process by retrieving the business meetings from the user's calendar program (e.g., Microsoft Outlook). Figure 1 shows the user interface of the *Travel Assistant* with the high level information for planning a trip to attend a business meeting. The interface displays a set of boxes showing values, which we call *slots*. A slot holds a current value and a set of possible values, which can be viewed in a pull-down list by clicking the arrow at the right edge of the slot. For example, there are slots for the subject and location of the meeting with values Travel Planner Meeting and DC respectively. The user could choose to plan another meeting from the list or input meeting information directly.

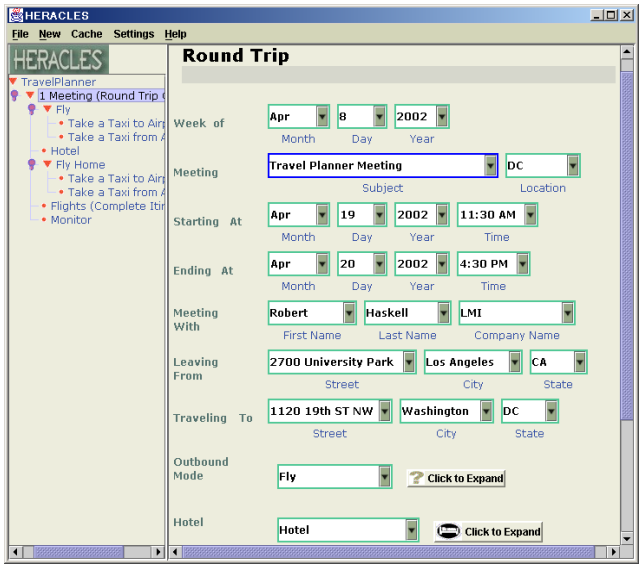

Figure 1: Planning a Meeting

Once the system has the details of the meeting, the next step is to determine how to get to the destination. There are three possible modes of transportation: Fly, Drive, or Take a Taxi. The system recommends the transportation mode based on the distance between the user's location and the meeting location. The system obtains the departure location from the user's personal profile and the meeting location from Outlook. The system computes the distance by first geocoding (determining the latitude and longitude) of the origin and destination addresses using the Mapblast Web site (www.mapblast.com). Then, using the geocoded information, a local constraint computes the distance between the two points. In our example, the distance between Los Angeles and Washington D.C. is 2,294 miles, so the system recommends that the user Fly. Since the meeting lasts for several days, it also recommends that the user stay at a hotel. Of course, the user can always override the system suggestions.

The *Travel Assistant* organizes the process of trip planning and the associated information hierarchically. The left pane of Figure 1 shows this hierarchical structure, with the particular choices made for the current plan. In this example, the trip consists of three tasks: flying to the meeting, staying at a hotel at the meeting location, and flying back home. Some tasks are further divided into subtasks, for example, how to get to and from the aiport when flying. In Heracles these tasks are represented by related slots and constraints that are encapsulated into units called *templates*, which are organized hierarchically.

The *Travel Assistant* helps the user evaluate tradeoffs that involve many different pieces of information and calculations. For example, Figure 2 illustrates how the system recommends the mode of transportation to the departure airport. This recommendation is made by comparing the cost of parking a car at the airport for the duration of the trip to the cost of taking a taxi to and from the airport. The system computes the cost of parking by retrieving the available airport parking lots and their daily rates from the AirWise site (www.airwise.com), determining the number of days the car will be parked based on scheduled meetings, and then calculating the total cost of parking. Similarly, the system computes the taxi fare by retrieving the distance between the user's home address and the departure airport from the Yahoo Map site (maps.yahoo.com), retrieving the taxi fare from the Washington Post site (www.washingtonpost.com), and then calculating the total cost. Initially, the system recomends taking a taxi since the taxi fare is only $19.50, while the cost of parking would be $48.00 using the Terminal Parking lot (the preferred parking lot in the user's profile). However, when the user changes the selected parking lot to Economy Lot B, which is $5 per day, this makes the total parking rate cheaper than the taxi fare, so the system changes the recommendation to Drive.

The system actively maintains the dependencies among slots so that changes to earlier decisions are propagated throughout the travel planning process. For example, Figure 3 shows how the Taxi template is affected when the user changes the departure airport in the higher-level Round Trip Flights template. In the original plan, the flight departs from Los Angeles International (LAX) at 11:55 PM. The user's preference is to arrive an hour before the departure time, thus he/she needs to arrive at LAX by 10:55 PM. Since Mapblast calculates that it takes 24 minutes to drive from the user's home to LAX, the system recommends leaving by 10:31 PM. When the user changes the departure airport from LAX to Long Beach airport (LGB), the system retrieves a new map and recomputes the driving time. Changing the departure airport also results in a different set of flights. The recommended flight from LGB departs at 9:50 PM and driving to LGB takes 28 minutes. Thus, to arrive at LGB by 8:50 PM, the system now suggests leaving home by 8:22 PM.

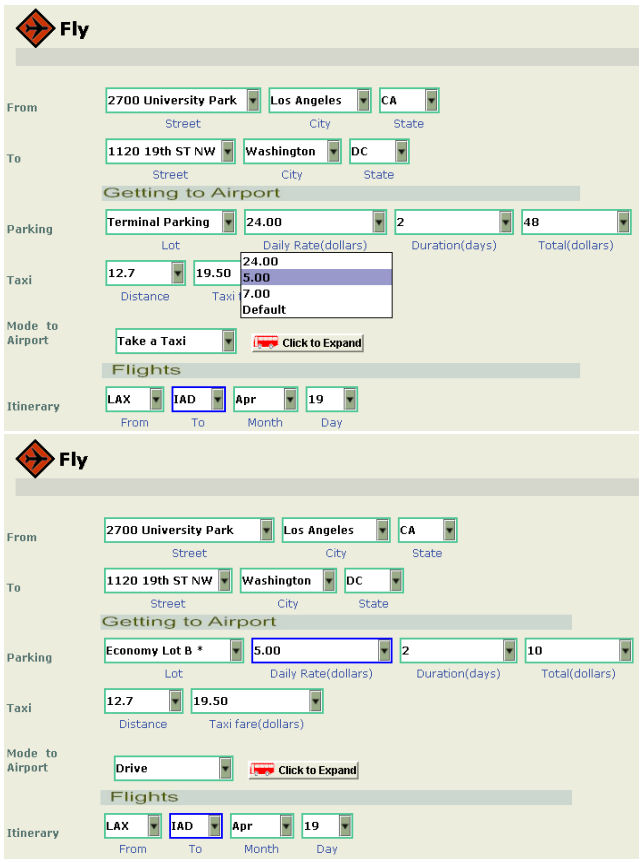

Figure 2: Comparing Cost of Driving versus Taking a Taxi

Monitoring Travel Status

There are various dynamic events that can affect a travel plan, for instance, flight delays, cancellations, fare reductions, etc. Many of these events can be detected in advance by monitoring information sources. The *Travel Assistant* is aware that some of the information it accesses is subject to change, so it delegates the task of following the evolution of such information to a set of monitoring agents. For instance, a flight schedule change is a critical event since it can have an effect not only on the user's schedule at the destination but also on the reservations at a hotel and a car rental agency. In addition to agents handling critical events, there are also monitoring agents whose purpose is to make a trip more convenient or cost-effective. For example, tracking airfares or finding restaurants near the current location of the user. In what follows, we describe the monitoring agents that we defined for travel planning. As shown in Figure 4, Heracles automatically generates the set of agents for monitoring a travel plan. Figure 5 shows some example messages sent by these agents.

The *Flight-Status* monitoring agent uses the ITN Flight Tracker site to retrieve the current status of a given flight. If the flight is on time, the agent sends the user a message to that effect two hours before departure. If the user's flight is delayed or canceled, it sends an alert through the user's

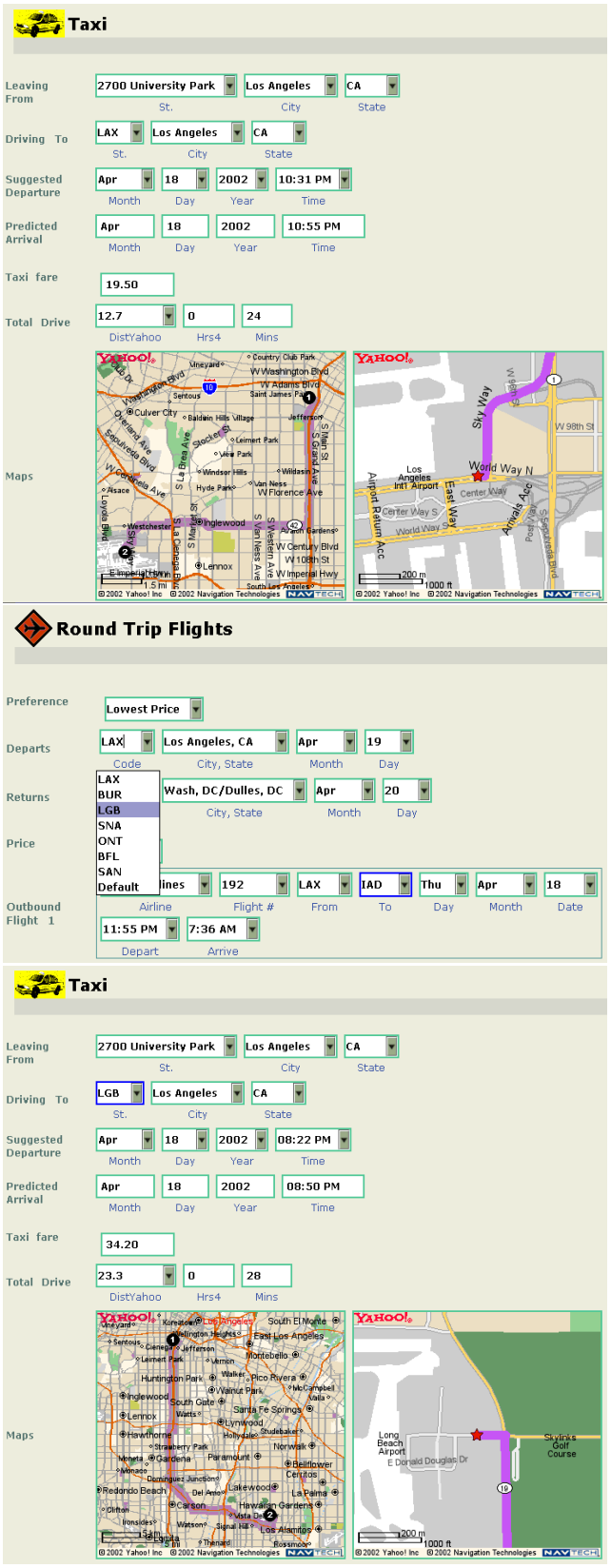

Figure 3: Change in Selected Airport Propagates to Drive Subtemplate

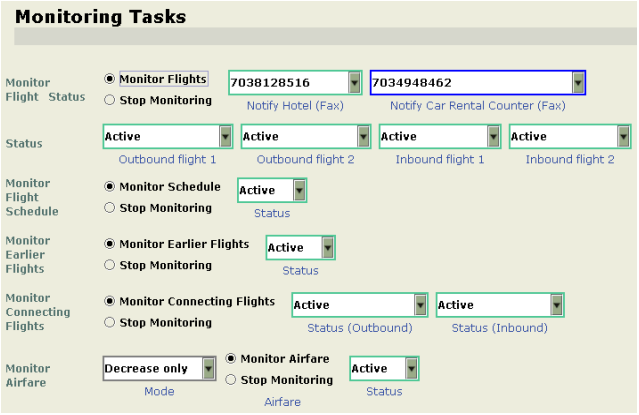

Figure 4: Template for Generating Monitoring Agents

(a) *Flight-Status Agent*: Flight delayed message

Your United Airlines flight 190 has been delayed. It was originally scheduled to depart at 11:45 AM and is now scheduled to depart at 12:30 PM. The new arrival time is 7:59 PM.

(b) *Flight-Status Agent*: Flight cancelled message

Your Delta Air Lines flight 200 has been cancelled.

(c) *Flight-Status Agent*: Fax to a hotel message

Attention : Registration Desk

I am sending this message on behalf of David Pynadath, who has a reservation at your hotel. David Pynadath is on United Airlines 190, which is now scheduled to arrive at IAD at 7:59 PM. Since the flight will be arriving late, I would like to request that you indicate this in the reservation so that the room is not given away.

(d) *Airfare Agent*: Airfare dropped message

The airfare for your American Airlines itinerary (IAD - LAX) dropped to $281.

(e) *Earlier-Flight Agent*: Earlier flights message

The status of your currently scheduled flight is:

190 LAX (11:45 AM) - IAD (7:29 PM) 45 minutes Late

The following United Airlines flight arrives earlier than your flight:

946 LAX (8:31 AM) - IAD (3:35 PM) 11 minutes Late

Figure 5: Actual Messages sent by Monitoring Agents

preferred device (e.g., a text message to a cellular phone). If the flight is delayed by more than an hour, the agent sends a fax to the car rental counter to confirm the user's reservation. If the flight is going to arrive at the destination airport after 5 PM, the agent sends another fax to the hotel so that the reserved room will not be given away. Since the probability of a change in the status of a flight is greater as the departure time gets closer, the agent monitors the status more frequently as the departure time nears.

The *Airfare* monitoring agent keeps track of current prices for the user's itinerary. The airlines change prices unpredictably, but the traveler can request a refund (for a fee) if the price drops below the purchase price. This agent gathers current fares from Orbitz (www.orbitz.com) and notifies the user if the price drops by more than a given threshold.

The *Flight-Schedule* monitoring agent keeps track of changes to the scheduled departure time of a flight (using Orbitz) and notifies the user if there is any change. Without such an agent, a traveler often only discovers this type of schedule changes after arriving at the airport.

The *Earlier-Flight* monitoring agent uses Orbitz to find the flights that leave earlier than the scheduled flight. It shows the alternative earlier flights and their status. This information becomes extremely useful when the scheduled flight is significantly delayed or canceled.

The *Flight-Connection* agent tracks the user's current flight, and a few minutes before it lands, it sends the user the gate and status of the connecting flight.

The *Restaurant-Finder* agent locates the user based on either a Global Positioning System (GPS) device or his/her expected location according to the plan. On request, it suggests the five closest restaurants provinding cuisine type, price, address, phone number, latitude, longitude, and distance from the user's location.

In the following sections, we describe in detail the technology we have used to automate travel planning and monitoring, namely, the Heracles interactive constraint-based planning framework and the Theseus agent execution and monitoring system.

Interactive Constraint-based Planning

The critical challenge for Heracles is integrating multiple information sources, programs, and constraints into a cohesive, effective tool. We saw examples of these diverse capabilities in the *Travel Assistant*, such as retrieving scheduling information from a calendar system, computing the duration of a given meeting, and invoking an information agent to find directions to the meeting.

Constraint reasoning technology offers a clean way to integrate multiple heterogeneous subsystems in a plug-and-play approach. Our approach employs a constraint representation where we model each piece of information as a distinct variable[1] and describe the relations that define the valid values of a set of variables as constraints. A constraint can be implemented either by a local procedure within the constraint engine or by an external component. In particular, we use information agents built with Theseus to implement many of the external constraints.

Using a constraint-based representation as the basis for control has the advantage that it is a declarative representation and can have many alternative execution paths. Thus, we need not commit to a specific order for executing components or propagating information. The constraint propagation system will determine the execution order in a natural manner. The constraint reasoning system propagates information entered by the user as well as the system's suggestions, decides when to launch information requests, evaluates constraints, and computes preferences.

[1] In the example in the previous section we have referred to each piece of information presented to the user as a slot. We use the term slot for user interface purposes. Each slot has a corresponding variable defined in the constraint network, but there may be variables that are not presented to the user.

Constraint Network Representation

A constraint network is a set of variables and constraints that interrelate and define the valid values for the variables. Figure 6 shows the fragment of the constraint network of the *Travel Assistant* that addresses the selection of the method of travel from the user's initial location to the airport. The choices under consideration are: driving one's car (which implies leaving it parked at the airport for the duration of the trip) or taking a taxi.

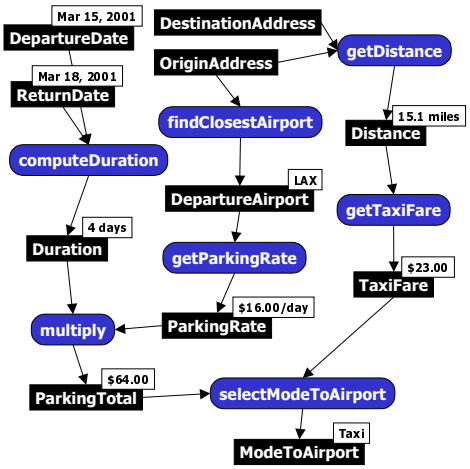

Figure 6: Constraint Network: Driving Versus Taking a Taxi

In the sample network of Figure 6, the variables are shown as dark rectangles and the assigned values as white rectangles next to them. The variables capture the relevant information for this task in the application domain, such as the DepartureDate, the Duration of the trip, the ParkingTotal (the total cost of parking for the duration of the trip), the TaxiFare, and the ModeToAirport. The DepartureAirport has an assigned value of LAX, which is assigned by the system since it is the closest airport to the user's address.

Conceptually, a *constraint* is a n-ary predicate that relates a set of *n* variables by defining the valid combinations of values for those variables. A constraint is a computable component which may be implemented by a local table look-up, by the computation of a local function, by retrieving a set of tuples from a remote information agent, or by calling an arbitrary external program. In Heracles the constraints are directed. The system evaluates a constraint only after it has assigned values to a subset of its variables, the *input* variables. The constraint evaluation produces the set of *possible values* for the *output* variables.

In the sample network of Figure 6 the constraints are shown as rounded rectangles. For example, the computeDuration constraint involves three variables (DepartureDate, ReturnDate, and Duration), and it's implemented by a function that computes the duration of a trip given the departure and return dates. The constraint getParkingRate is implemented by calling an information agent that accesses a web site that contains parking rates for airports in the US.

Each variable can also be associated with a preference constraint. Evaluating the preference constraint over the possible values produces the *assigned* value of the variable. (The user can manually reset the assigned value at any point by selecting a different alternative.) Preference constraints are often implemented as functions that impose an ordering on the values of a domain. An example of a preference for business travel is to choose a hotel closest to the meeting.

Constraint Propagation

Since the constraints are directed, the constraint network can be seen as a directed graph. In the current version of the system, this constraint graph must be acyclic, which means that information flows in one direction. This directionality simplifies the interaction with the user. Note that if the user changes a variable's value, this change may need to be propagated throughout the constraint graph.

The constraint propagation algorithm proceeds as follows. When a given variable is assigned a value, either directly by the user or by the system, the algorithm recomputes the possible value sets and assigned values of all its dependent variables. This process continues recursively until there are no more changes in the network. More specifically, when a variable X changes its value, the constraints that have X as input variable are recomputed. This may generate a new set of possible values for each dependent variable Y. If this set changes, the preference constraint for Y is evaluated selecting one of the possible values as the new assigned value for Y. If this assigned value is different from the previous one, it causes the system to recompute the values for further downstream variables. Values that have been assigned by the user are always preferred as long as they are consistent with the constraints.

Consider again the sample constraint network in Figure 6. First, the constraint that finds the closest airport to the user's home address assigns the value LAX to the variable DepartureAirport. Then, the constraint getParkingRate, which is a call to an information agent, fires producing a set of rates for different parking lots. The preference constraint selects terminal parking which is $16.00/day. This value is multiplied by the duration of the trip to compute the ParkingTotal of $64 (using the simple local constraint multiply). A similar chain of events results in the computation of the TaxiFare. Once the ParkingTotal and the TaxiFare are computed, the selectModeToAirport constraint compares the costs and chooses the cheapest means of transportation, which in this case is to take a Taxi.

Template Representation

As described previously, to modularize an application and deal with its complexity, the user interface presents the planning application as a hierarchy of templates. For example, the top-level template of the *Travel Assistant* (shown in Figure 1) includes a set of slots associated with who you are meeting with, when the meeting will occur, and where the meeting will be held. The templates are organized hierarchically so that a higher-level template representing an abstract task (e.g., Trip) may be decomposed into a set of more specific subtasks, called subtemplates (e.g., Fly, Drive, etc). This hierarchical task network structure helps to manage the

complexity of the application for the user by hiding less important details until the major decisions have been achieved.

This template-oriented organization has ramifications for the constraint network. The network is effectively divided into partitions, where each partition consists of the variables and constraints that compose a single template. During the planning process the system only propagates changes to variables within the template that the user is currently working on. This strategy considerably improves performance.

Information Agents

Our system uses information agents to support both the trip planning and monitoring. While information agents are similar to other types of software agents, their plans are distinguished by a focus on gathering, integrating, and monitoring of data from distributed and remote sources. To efficiently perform these tasks we use Theseus (Barish *et al.* 2000; Barish & Knoblock 2002), which is a streaming dataflow architecture for plan execution. In this section, we describe how we use Theseus to build agents capable of efficiently gathering and monitoring information from remote data sources.

Defining an Information Agent

Building an information agent requires defining a plan in Theseus. Each plan consists of a name, a set of input and output variables, and a network of operators connected in a producer-consumer fashion. For example, the *Flight-Status* agent takes flight data (i.e., airline, flight number) as input and produces status information (i.e., projected arrival time) as output.

Each operator in a plan receives input data, processes it in some manner, and outputs the result - which is then potentially used by another operator. Operators logically process and transmit data in terms of relations, which are composed of a set of tuples. Each tuple consists of a set of attributes, where an attribute of a relation can be either a string, number, nested relation, or XML object.

The set of operators in Theseus support a range of capabilities. First, there are information gathering operators, which retrieve data from local and remote sources including web sites. Second, there are data manipulation operators, which provide the standard relational operations, such as select, project and join, as well as XML manipulation operations using *XQuery*. Third, there are monitoring-related operators, which provide scheduling and unscheduling of tasks and communication with a user through email or fax.

Plans in Theseus are just like operators: they are named and have input and output arguments. Consequently, any plan can be called as an operator from within any other plan. This subplan capability allows a developer to define new agents by composing existing ones.

Accessing Web Sources

Access to on-line data sources is a critical component of our information agents. In the *Travel Assistant* there is no data stored locally in the system. Instead, all information is accessed directly from web sources. To do this we build *wrappers* that enable web sources to be queried as structured data sources. This makes it easy for the system to manipulate the resulting data as well as integrate it with information from other data sources.

For example, a wrapper for Yahoo Weather dynamically turns the source into XML data in response to a query. Since the weather data changes frequently, it would not be practical to download this data in advance and cache it for future queries. Instead, the wrapper provides access to the live data, but provides it in a structured form.

We have developed a set of tools for semi-automatically creating wrappers for web sources (Knoblock *et al.* 2000). The tools allow a user to specify by example what the wrapper should extract from a source. The examples are then fed to an inductive learning system that generates a set of rules for extracting the required data from a site.

Once a wrapper for a site has been created, Theseus agents can programmatically access data from that site using the wrapper operator in their plans. For example, with the wrapper for Yahoo Weather, we can now send a request to get the weather for a particular city and it will return the corresponding data.

Monitoring Sources

In addition to being able to gather data from web sources, Theseus agents are capable of *monitoring* those sources and performing a set of actions based on observed changes. The monitoring is performed by retrieving data from online sources and comparing the returned results against information that was previously retreived and stored locally in a database. This provides the capability to not only check current status information (e.g., flight status), but to also determine how the information has changed over time.

There are several ways in which plans can react to a monitoring event. First, a plan can use e-mail or fax operators to asynchronously notify interested parties about important updates to monitored data. Second, a plan can schedule or unschedule other agents in response to some condition. For example, once a flight has departed, the flight monitoring agent can schedule the *Flight-Connection* agent to run a few minutes before the scheduled arrival time.

Theseus agents are managed by a scheduling system that allows agents to be run once or at a fixed interval. The agent scheduler works by maintaining a database of the tasks to run and when they are scheduled to run next. Once scheduled, agents are launched at the appropriate time. Once they are run, the database is updated to reflect the next time the task is to be run. Since the Theseus plan language supports operators that can schedule and unschedule agents, this provides the ability to run new agents at appropriate times based on events in the world.

As an example information agent that monitors a data source, consider the plan for the *Flight-Status* agent shown in Figure 7. Initially, the agent executes a wrapper operator to retrieve the current flight status information. It then uses another online source to normalize the information based on the time zone of the recipient. The resulting normalized flight status information indicates that the flight is either arrived, cancelled, departed, or pending (waiting to depart). If the flight has been cancelled, the user is notified via the email

operator. In this case, the flight needs no additional monitoring and the unschedule operator is used to remove it from the list of monitored flights. For departed flights the *Flight-Connection* agent is scheduled. For each pending flight, the agent must perform two actions. First, it checks if the arrival time of the flight is later than 5pm and if so it uses the fax operator to notify the hotel (it only does this once). Second, the agent compares the current departure time with the previously retrieved departure time. If they differ by a given threshold, the agent does three things: (a) it faxes a message to the car rental agency to notify them of the delay, (b) it updates its local database with the new departure time (to track future changes) and (c) it e-mails the user.

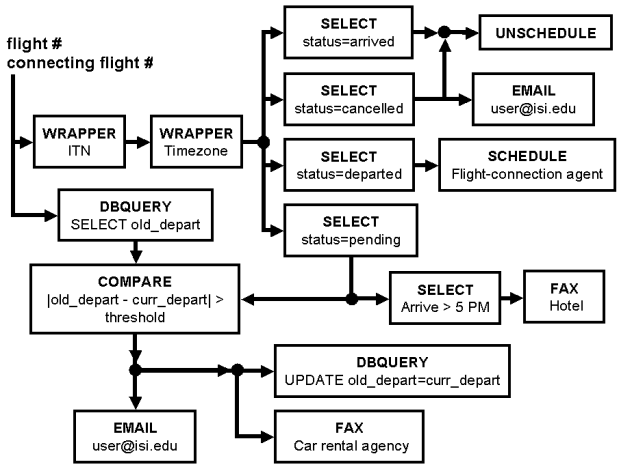

Figure 7: The Flight Status information monitoring agent

Efficiently Executing Agent Plans

Information agent plans are unique in two key respects. First, they tend to integrate data from multiple sources - for example, the *Flight-Status* agent might query multiple Internet real-time data sources to find out the in-flight status of a particular airplane. Second, they usually gather and monitor data from sources that are remote and deliver unpredictable performance - such as Internet web sites. Thus, information agent execution is often I/O-bound - with an agent spending the majority of its execution time waiting for replies from remote sources it has queried.

To efficiently execute plans that primarily integrate data from remote sources, Theseus employs a dataflow model of execution. Under this model, plan operators are arranged in a producer-consumer fashion, leading to partially ordered plans. Then, at run-time, operators can execute in parallel when their individual inputs have been received. This decentralized form of execution maximizes the amount of *horizontal parallelism*, or concurrency between independent data flows, available at run-time.

Theseus alo supports the *streaming* of data between plan operators. Streaming enables consumer operators to receive data emitted by their producers as soon as possible. This, in turn, enables these consumers to process this data immediately and communicate output to other consumers farther downstream as soon as possible. For example, consider an agent plan that fetches a large amount of data from two slow sources and joins this information together. Streaming enables the join to be executed as soon as possible on the available data, as it trickles in from either source. As a result, with streaming, producers and consumers may be processing the same logical set of data concurrently, resulting in a form of pipelined or *vertical parallelism* during execution.

Related Work

Most commercial systems for travel planning take the traditional approach of providing tools for selecting flights, hotel, and car rentals in separate steps. The only integrated approach is a system called MyTrip from XTRA On-line. Based on personal calendar information, the system automatically produces a complete plan that includes the flights, hotel and car rental. Once it has produced a plan, the user can then edit the individual selections made by the system. Unlike the *Travel Assistant*, the user cannot interactively modify the plan, such as constraining the airlines or departure airport. Also, MyTrip is limited to only the selection of flights, hotels, and car rentals. In addition to MyTrip, there exist some commercial systems (such as the one run by United Airlines) that provide basic flight status and notification. However, these systems do not actually track changes in the flight status over time (they merely notify passengers a fixed number of hours before flights) and they do not notify hotels about flight delays or suggest earlier flights or better connections when unexpected events (e.g., bad weather) occur.

In terms of constraint reasoning, there is a lot of research on constraint programming (Saraswat & van Hentenryck 1995), but not much attention has been paid to the interplay between information gathering and constraint propagation and reasoning. Bressan and Goh (1997) have applied constraint reasoning technology to information integration. They use constraint logic programming to find relevant sources and construct an evaluation plan for a user query. In our system, the relevant sources have already been identified. In dynamic constraint satisfaction (Mittal & Falkenhainer 1990), the variables and constraints present in the network are allowed to change with time. In our framework, this is related to interleaving the constraint satisfaction with the information gathering. Lamma et al. (1999) propose a framework for interactive constraint satisfaction problems (ICSP) in which the acquisition of values for each variable is interleaved with the enforcement of constraints. The information gathering stage in our constraint integration framework can also be seen as a form of ICSP. Their application domain is on visual object recognition, while our focus is on information integration

Our work on Theseus is related to two existing types of systems: *general agent executors* and *network query engines*. General agent executors, like RAPS (Firby 1994) and PRS-Lite (Myers 1996) propose highly concurrent execution models. The dataflow aspect of Theseus can be seen as another such model. The main difference is that execution in Theseus not only involves enabling operators but routing

information between them as well. In this respect, Theseus shares much in common with several recently proposed network query engines (Ives *et al.* 1999; Hellerstein *et al.* 2000; Naughton *et al.* 2001). Like Theseus, these systems focus in efficiently integrating web-based data. However, network query engines have primarily focused on performance issues; while Theseus also acts as an efficient executor, it is distinguished from these other query engines by (a) its novel operators that facilitate monitoring and asynchronous notification and (b) its modular dataflow agent language that allows a wider variety of plans to be built and executed.

Discussion

The travel planning and monitoring are fully functional systems that are in use today. The planner is not yet directly connected to a reservation system, but it is a very useful tool for helping to make the myriad of decisions required for planning a trip. Likewise, the monitoring agents are able to continually monitor all aspects of a trip and provide immediate notification of changes and cancellations. There are existing commercial systems that provide small pieces of these various capabilities, but the technology and application presented here is unique in providing a complete end-to-end solution that plans and then monitors all aspects of a trip. For example, the planning process will automatically produce the fax numbers required for use by the monitoring agents to notify the hotel and car rental agency. The current system is in use at the Information Sciences Institute and we plan to make this system more widely available on the Web as part of the Electric Elves Project (Chalupsky *et al.* 2001).

One limitation of the current system is that the monitoring agents do not communicate problems or changes back to the travel planner. Ideally, if a flight is canceled one would want the system to re-book the traveler on another flight as soon as possible and then make any other needed changes to the itinerary based on the changes to the flight. Or if the price of a ticket declines to automatically rebook the ticket. The current system does not do this. We are working on the next generation of the travel planning component which will support this type of dynamic replanning of a trip based on changes in the world.

Acknowledgments

The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197, F30602-00-1-0504, F30602-98-2-0109, in part by the Air Force Office of Scientific Research under grant number F49620-01-1-0053, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, cooperative agreement number EEC-9529152. The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Barish, G., and Knoblock, C. A. 2002. Speculative plan execution for information gathering. In *Proceedings of the 6th International Conf. on AI Planning and Scheduling*.

Barish, G.; DiPasquo, D.; Knoblock, C. A.; and Minton, S. 2000. A dataflow approach to agent-based information management. In *Proceedings of the 2000 International Conference on Artificial Intelligence (IC-AI 2000)*.

Bressan, S., and Goh, C. H. 1997. Semantic integration of disparate information sources over the internet using constraint propagation. In *Workshop on Constraint Reasoning on the Internet (at CP97)*.

Chalupsky, H.; Gil, Y.; Knoblock, C. A.; Lerman, K.; Oh, J.; Pynadath, D. V.; Russ, T. A.; and Tambe, M. 2001. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Thirteenth Innovative Applications of Artificial Intelligence Conference*.

Firby, R. J. 1994. Task networks for controlling continuous processes. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*.

Hellerstein, J. M.; Franklin, M. J.; Chandrasekaran, S.; Deshpande, A.; Hildrum, K.; Madden, S.; Raman, V.; and Shah, M. A. 2000. Adaptive query processing: technology in evolution. *IEEE Data Engineering Bulletin* 23(2):7–18.

Ives, Z. G.; Florescu, D.; Friedman, M.; Levy, A.; and Weld, D. S. 1999. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Knoblock, C. A.; Lerman, K.; Minton, S.; and Muslea, I. 2000. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin* 23(4).

Knoblock, C. A.; Minton, S.; Ambite, J. L.; Muslea, M.; Oh, J.; and Frank, M. 2001. Mixed-initiative, multi-source information assistants. In *Proceedings of the Tenth International World Wide Web Conference*.

Lamma, E.; Mello, P.; Milano, M.; Cucchiara, R.; Gavanelli, M.; and Piccardi, M. 1999. Constraint propagation and value acquisition: Why we should do it interactively. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 25–32.

Myers, K. L. 1996. A procedural knowledge approach to task-level control. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*.

Naughton, J. F.; DeWitt, D. J.; Maier., D.; et al. 2001. The niagara internet query system. *IEEE Data Engineering Bulletin* 24(2):27–33.

Saraswat, V., and van Hentenryck, P., eds. 1995. *Principles and Practice of Constraint Programming*. Cambridge, MA: MIT Press.