Integrating Planning and Execution for Information Gathering*

Craig A. Knoblock
Information Sciences Institute and Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
Email: knoblock@isi.edu

Abstract

Current specialized planners for query processing are designed to work in local, reliable, and predictable environments. However, there are a number of problems that arise in gathering information from large networks of distributed information. In this environment actions can be executed in parallel to exploit distributed resources, new goals come into the system in the midst of execution, actions may fail due to exogenous events and need to be replanned, and there is incomplete information about the world. We have developed a planner called Sage that addresses the issues that arise in this environment. This system integrates previous work on planning, execution, replanning, and sensing and extends this work to support simultaneous and interleaved planning and execution. Sage has been applied to the problem of information gathering to provide a flexible and efficient system for integrating heterogeneous and distributed data and has been used in the domains of logistics planning and trauma care.

Introduction

The task of information gathering requires locating, retrieving, and integrating information from large numbers of distributed and heterogeneous information sources. In this environment flexibility and efficiency are critical. The usual approach to generating a plan for processing information and then executing it is inflexible and may be very inefficient if problems arise in the query processing. The problem is that actions may fail, the system has incomplete information about the environment, new goals may arise at any time, and the overall efficiency is important.

*The research reported here was supported in part by Rome Laboratory of the Air Force Systems Command and the Advanced Research Projects Agency under contract no. F30602-91-C-0081, and in part by the National Science Foundation under grant number IRI-9313993. The views and conclusions contained in this report are those of the authors and should not be interpreted as representing the official opinion or policy of RL, ARPA, NSF, the U.S. Government, or any person or agency connected with them.

To address these problems, we have developed a planning system that builds on the previous work on planning, execution, sensing, and replanning. The planner, which we call Sage, was implemented by augmenting UCPOP (Penberthy & Weld 1992, Barrett *et al.* 1993) with the capabilities to produce parallel execution plans (Knoblock 1994, Wilkins 1984), interleave planning and execution (Ambros-Ingerson 1987, Etzioni, Golden, & Weld 1994), support run-time variables (Ambros-Ingerson 1987, Etzioni *et al.* 1992), and perform replanning where appropriate (Hanks & Weld 1992). We have integrated all of these capabilities into a single unified system in which the actions can be executed simultaneously, and the planning, sensing, and replanning can be performed during execution. This allows the system to execute operations in parallel, augment and replan portions of the plan that is currently being executed, and receive and plan new tasks within the context of the current plan.

The contributions of this work are twofold. First, we extend the previous work by tightly integrating these components and adding the capability to execute actions in parallel with the planning, replanning, and sensing. Second, we show how the resulting planner can be effectively applied to the real-world problem of information gathering from distributed and heterogeneous information sources.

The paper is organized by presenting the individual components of the planner and describing how each component supports the task of information gathering. The second section describes the basic planning system and how it supports planning for parallel actions. The third section presents our approach to tightly integrating the planning and execution and how the actions are performed simultaneously. The fourth section shows how the planner handles failures by replanning. The fifth section describes the support for sensing, which allows the system to gathering additional information at run time. The sixth section discusses the related work on both planning and information gathering. Finally, we conclude with a discussion of the contributions and directions for future work.

Planning in Sage

We have implemented Sage on top of the UCPOP partial-order planner. UCPOP provides an expressive operator language that includes conjunction, negation, disjunction, existential and universal quantifiers, conditional effects, and a functional interface that allows preconditions to be implemented as Lisp functions. In this section we describe the extensions to the basic planner to produce parallel execution plans and then describe how the information gathering task is cast as a planning problem.

Parallel Execution Plans

Partial-order planners, such as UCPOP, produce plans with actions that are unordered, but if two actions are left unordered it only means that they can be executed in either order. To execute actions in parallel in a classical planner requires that two additional conditions be met. First, the simultaneous execution of two actions cannot change the outcome of the individual actions. Second, any potential resource conflicts must be captured in the representation of the operators in order to avoid conflicts during execution. To address the first issue, we will simply assume that executing any of the actions in parallel does not change the outcome.[1] The second issue cannot be ignored so easily since resource conflicts do arise in many problem domains, including information gathering.

Most resource conflicts can be represented in the preconditions and effects and resolved in the process of ensuring that the preconditions and effects are consistent. However, because of the limited representation, the type of conflict that is not typically handled in a partial-order planner is when two actions require the same reusable resource. This type of resource conflict is not captured by the preconditions and effects because at the start of execution the resource is available and when execution completes it is available again. In terms of the basic planning in Sage, the changes to UCPOP that were required were to add explicit resource definitions to the operators, modify the planner to enforce the resource constraints, and construct an evaluation function to exploit the use of parallelism in constructing plans.

The resource requirements of the operators are made explicit by augmenting each operator with a resource declaration. This is similar to the way resources are declared in Sipe (Wilkins 1984), but as described below, the way they are handled is different. An example operator with a resource declaration is shown in Figure 1. This operator describes the action of moving data from one data source to another and declares the data server from which the data is being moved as a resource. The purpose of this declaration is to prevent one operator from being executed in parallel with another operator that requires the same database server.

```
(define (operator move)
   :parameters (?db1 ?server1 ?db2 ?server2 ?data)
   :resources ((resource ?server1 server))
   :precondition (:and (available ?db1 ?server1 ?data)
                       (:neq ?db1 ?db2))
   :effect (:and (:not (available ?db1 ?server1 ?data))
                 (available ?db2 ?server2 ?data)))
```

Figure 1: Operator with Resource Declaration

In order to avoid resource conflicts, we modified the planner to ensure that if two operators require the same resource, then they are not left unordered relative to one another. In SIPE this is done with a critic that is run once at each planning level and checks for resource conflicts and then imposes ordering constraints when conflicts are found. In SAGE, every time a new action is added to the plan, the planner checks for potential resource conflicts with any other operator that is unordered with respect to the new action. Any conflicts discovered are added to the list of threats that must be removed before the plan is considered complete. These conflicts can be resolved immediately or delayed until later in the planning process. The default is to resolve the threats immediately.

Since efficiency is the primary motivation for generating parallel plans, we constructed an evaluation function that can be used to find plans with low overall parallel execution time. We assume we are given a domain-specific evalation for estimating the cost of each individual action. Using this action evaluation function, we can then implement a plan evaluation function that takes the parallelism into account to determine the overall execution cost. This is done using a dynamic-programming algorithm that recursively assigns a cost to the execution of the partial plan up to and including the action at the given node. The cost is calculated by adding the cost of the action at the node to the maximum cost of all the immediately prior partial plans. Once the cost of the partial-plan plan up to a node has been computed, we store this value so it will only need to be calculated once. Since each node (n) and each edge (e) in the graph is visited only once, the complexity of evaluating the plan cost is O(max(n,e)).

Planning for Information Gathering

In this section we describe how the information gathering task is cast as a planning problem in Sage. This problem requires developing an ordered set of operations for generating a requested set of data. This includes selecting the sources for the data, the operations for processing the data, the sites where the operations

[1] A more complete treatment of this problem requires a more expressive representation of time in order to capture the subtleties of different ways in which operators can interact. These capabilities are provided by temporal planners and such planners would be more appropriate for domains involving these types of interactions (Allen *et al.* 1991). Of course, this capability comes with a higher cost for reasoning about plans.

will be performed and the order in which to perform them. Since data can be moved around between different sites, processed at different locations, and the operations can be performed in a variety of orders, the space of possible plans is large. In this section we will describe the representation of the problem as a planning problem and describe how our planner is used to solve this problem.

The goal of an information gathering problem consists of a description of a set of desired data as well as the location where that data is to be sent. For example, Figure 2 illustrates a goal which specifies that the set of data be sent to the OUTPUT device of the SIMS information mediator (Arens *et al.* 1993, Knoblock, Arens, & Hsu 1994). The third argument of this goal specifies the data to be retrieved and is defined using the syntax of the query language of the Loom knowledge representation system (MacGregor 1990). This particular query requests all port names of seaports that are sufficiently deep to accommodate "breakbulk" ships.

```
(available output sims
      (retrieve (?port-name)
       (:and (seaport ?sport)
             (primary-port-name ?sport ?port-name)
             (geoloc-code ?sport ?glc-code)
             (channel ?channel)
             (geoloc-code ?channel ?glc-code)
             (channel-depth ?channel ?depth)
             (transport-ship ?ship)
             (vehicle-type-name ?ship "breakbulk")
             (ship-max-draft ?ship ?draft)
             (< ?draft ?depth)))
```

Figure 2: Example Information Gathering Goal

The initial state of a problem defines the information servers that are available as well as which server provides which information sources. The example shown in Figure 3 defines two servers, an Oracle database server running on an HP workstation, called **hp-oracle**, and an another Oracle server running on a Sun workstation, called **sun-oracle**. Both servers contain identical copies of the GEO and ASSETS databases. In addition to this information, there is also knowledge about the contents of the information sources that is stored in a Loom knowledge base. However, this information is static and is accessed directly through the functional interface rather than through the literals listed in the initial state.

```
((server-available hp-oracle)
 (server-available sun-oracle)
 (server geo hp-oracle)
 (server assets hp-oracle)
 (server geo sun-oracle)
 (server assets sun-oracle))
```

Figure 3: Example Initial State

There are twelve general operators used to plan out the processing of a query. These operators include: a **move** operator for moving a set of data from one information source to another, a **join** operator that combines two sets of data into a combined set of data, and **retrieve** operator for selecting the information source for retrieving a set of data. Each of the operators is instantiated at planning time with the particular set of data being manipulated as well as the database where the manipulation is being performed.

Consider the operator shown in Figure 4 that defines a join performed in the local system. (Note that the system only reasons about joins performed in the local system since any join to be performed in a remote database is specified implicitly by the query sent to that database.) This operator is used to achieve the goal of making some information available in the local knowledge base of the SIMS server. It does this by partitioning the request into two subsets of the requested data, getting that information into the local system and then joining that data together to produce the requested set of data. The **available** preconditions are achieved by other operators and the **join-partition** precondition is defined by a program that produces the relevant partitions of the requested data.

```
(define (operator join)
  :parameters (?join-ops ?data ?data-a ?data-b)
  :precondition (:and (join-partition ?data ?join-ops ?data-a ?data-b)
                      (available local sims ?data-a)
                      (available local sims ?data-b))
  :effect (available local sims ?data))
```

Figure 4: The Join Operator

The plan generated for the example query in Figure 2 is shown in Figure 5. In this example, the system partitions the given query such that the ship information is retrieved in a single query to the ASSETS information source and the seaport and channel information is retrieved in a single query to the GEO information source. The planner leaves the join between the seaports and channels to be performed by the remote GEO information server since this will be cheaper than moving the information into the local system. If the system could perform all of the work in one remote system, then it would completely bypass the local system and send the data directly to the output. All of the information is brought into the local SIMS server where the draft of the ships can be compared against the depth of the seaports. Once the final set of data has been generated, it is sent to the output.

To search the space of query access plans efficiently, the system uses a set of domain-specific heuristics and an evaluation function for estimating the cost of the various operations. The planner uses the evaluation function in a branch-and-bound search to produce a low cost parallel execution plan. The evaluation function allows the different partial plans to be compared and partial plans that are more expensive than the final plan will not need to be expanded further. The domain-specific heuristics help prune the search space

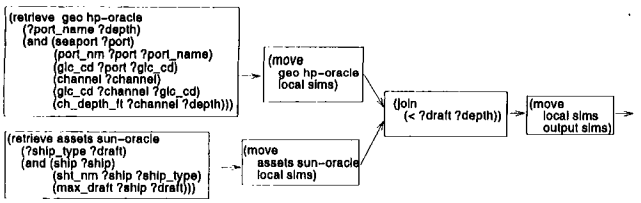

Figure 5: Parallel Query Access Plan

by eliminating plans that are unlikely to be cheaper than other plans. For example, a standard heuristic is to avoid considering all possible join permutations. Both the heuristics and the fact that the evaluation function only produces estimates may prevent the planner from finding the optimal plan. However, the goal is to produce high quality and not optimal plans.

This approach to searching the space of plans is similar to what is done in other systems for producing query plans for relational databases (Selinger *et al.* 1988). These systems typically generate the space of query access plans, constraining the space of plans with appropriate domain-specific heuristics, and then evaluate the plans and select the best one. In the following sections we will describe functionality that goes beyond what existing database systems provide.

Execution in Sage

The execution of the actions in a plan is tightly integrated with the planning process. This capability combined with the parallel execution allows the system to support replanning of failed actions, plan for new goals as they arrive, and exploit sensing operations, all while the system is executing other actions in a plan. In this section we describe the integration of planning and execution and its use for information gathering.

Integrating the Planning and Execution

The planning and execution is integrated by considering the execution as an integral part of the planning process. This is done by treating the execution of each individual action as a necessary step in completing a plan. The goal of the planner becomes producing a complete and executed plan rather than just producing a complete plan. Just as achieving all of the preconditions of a plan are required for a complete plan, executing each of the actions is also part of the final result.

The underlying planner, UCPOP, maintains a list of flaws, which are the agenda of things that need to be done to complete a particular plan. These flaws include open conditions, which are the preconditions that have not yet been achieved, and threats, which are potential interactions between operators that must be resolved by adding ordering or binding constraints. We integrated the execution in Sage by adding two new types of flaws: an *unexecuted* action and an *executing* action. Whenever a new operator is added to a plan, the corresponding flaw indicating that the action is unexecuted is also added to the agenda. This flaw can only be worked on if there are no open conditions for the operator and every operator that achieves a precondition of the operator has already been executed. If any of these conditions are not already met, then there will be at least one other flaw for the system to work on first.

The execution of an action is viewed as a commitment to the plan in which the action occurs. This means that the planner will only consider the plan from which the action is executed and all valid refinements of that plan. The reason for this is to avoid executing actions from two different, possibly inconsistent, plans. Since execution of an action commits to the corresponding plan, we would like the planner to be selective in choosing to execute an action. This is achieved by delaying work on any *unexecuted* flaw for as long as possible. The idea is that the planner should find the best complete plan before any action is executed. Then once execution is begun, it would resolve any failed subplans or new goals before executing the next action. This means that the planner will never execute an action until there are no better choices to consider (relative to the evaluation function).

Since the execution of an action may take considerable time, we do not want the system to execute an action and wait for the results. Instead, the planner spawns a new process that executes the action and notifies the planner once it has completed. To keep track of the actions currently being executed, the corresponding unexecuted flaw is removed from the agenda and a new flaw is added indicating that the action is currently executing. At any one time there may be a number of actions that are all executing simultaneously. On each cycle of the planner, the system will check if any executing actions have completed. Once a action is completed then the *executing* flaw is removed from the agenda. If it completes successfully then the action is left in the plan and marked as completed. Other actions that depend on this action may now be executable if all of their other dependencies have also been executed. If an action fails, the replanning module is invoked, which is described in the next section.

The ability to plan in parallel with the execution means that the planner can run continuously, replanning failed actions, perform sensing operations, and accepting new goals and planning for them while it is executing other actions. The system first checks for open conditions that need to be achieved and threats that need to be resolved. Next, it checks for actions that have completed execution and updates the plan according to the outcome. Then, it checks for any new goal conditions and if any are found on the queue they are added to the open conditions of the currently exe-

cuting plan. Finally, the system initiates execution of any operators for which the preceding operators have already completed successfully. In every case, the system does the planning in the context of the current executing plan and if an action is executed, an action terminates, or a new action is added, the correspond plan becomes the current plan.

Execution and Information Gathering

The integration of the execution and the planning provides for a much more flexible system. It allows the system to plan for new goals as they arise, replan failed actions, and interleave sensing operations. The replanning and sensing are described in the next two sections. In the rest of this section we describe how the system incorporates a new goal into an already executing plan.

Consider what would happen if a new goal is given to the system while it is executing the plan in Figure 5. Assume that the system has already executed some of the actions and it is in the midst of executing others, as shown in Figure 6. When a new goal comes in of retrieving the description of the Long Beach seaport, the planner notices the pending goal on the next cycle and then searches for the appropriate additions to the currently executing plan to solve this goal. While the system is generating this plan, the action in progress (shown by the action in the box with thick lines) will continue to execute since actions are run as separate processes.

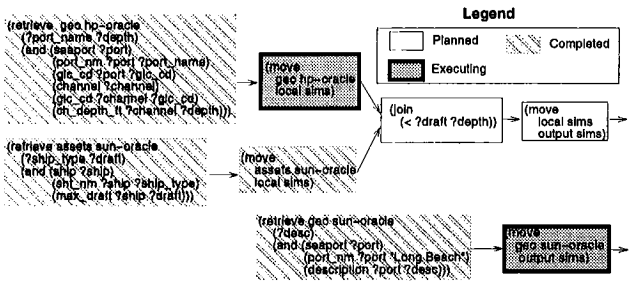

Figure 6: Simultaneous Planning and Execution

The resulting plan is shown in Figure 6. The advantage of planning this new goal in the context of the existing plan is that shared work can be exploited and any potential resource conflicts are considered in the planning process. In this case, the goal requires access to the **geo** database, which is already in use by the other executing query. As a result, the system uses the **geo** database running on the **sun-oracle** server since the other action that required this resource has already completed. The separate top-level goals are treated as independent goals, so if a subplan fails it will not cause unrelated goals to fail. In addition, as soon as any top-level goal is complete, the results are sent to the calling process. This allows the planner to run continuously and return results as soon as they are obtained rather than waiting for a plan to complete.

Replanning in Sage

A problem that arises in executing actions in the world are that the actions may fail. Ideally, when an action fails we would like to avoid throwing out the entire plan and starting from scratch. There may already be considerable effort expended in execution of the plan so far. Instead, we would like to replan the failed portion of the plan, while maintaining as much of the executing plan as possible. The replanning in Sage is based on the approach used in the Systematic Plan Adaptor (SPA) (Hanks & Weld 1992), which systematically searches the space of plan modifications. The basic idea is to annotate the plans that are being considered to avoid generating either the plan that failed or other redundant plans. Within the Sage framework, this replanning can be performed while other actions are still executing.

Replanning for Information Gathering

In the information gathering task, the ability to replan upon failure can be exploited to handle failures by redirecting a query to a different information source. Actions may fail in at least two ways. First, an error in the execution may occur because the database is down, the network is down, or an improperly formulated query was sent to the database. In this case the failure is easy to recognize since an error is returned from the process executing the action. Second, a query may not return the expected data. This case is a bit harder to detect since it requires formulating expectations about what data should be returned and then comparing the actual data with the expectations. We currently identify the first case of failure and we are working on the second.

An example of a failed action that can be replanned is shown in Figure 7. The actions in the dashed boxes are the failed actions and the actions above the failed ones are the replanned actions. Since the replanned **move** action requires the same resource as the action currently being executed, an ordering constraint is added between these two actions. Note that the replanning takes place while the other action continues to execute.

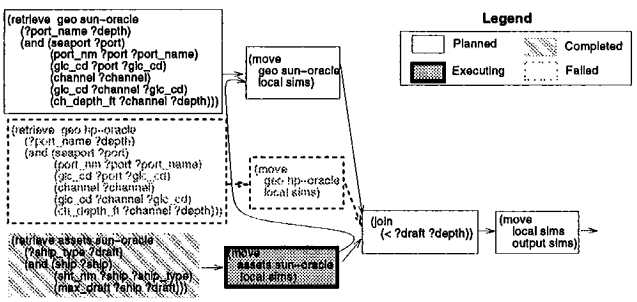

Figure 7: Replanning a Failed Plan

Sensing in Sage

Sensing allows the planner to gather intermediate data to aide the processing. Based on earlier work (Ambros-Ingerson 1987, Etzioni *et al.* 1992), we have added support for run-time variables, which allows the planner to perform sensing operations in the course of planning. In the context of information gathering, knowing some specific information stored in an information source can be useful in formulating a query to another information source.

Run-time variables appear in the effects of operators and essentially serve as place holders for the value or values returned by the action at the point it is executed. These variables are useful because the result can be incorporated and used in other parts of the plan. An issue that arises in the use of run-time variables is that until desired information is available, the planning may have to be postponed or a plan with all possible contingencies will have to be produced in order to deal with the possible returned values. Similar to some of the other systems that perform sensing (i.e., IPEM (Ambros-Ingerson 1987) and UWL (Etzioni *et al.* 1992)), Sage currently delays working on any open condition that involves a run-time variable. In the remainder of this section we describe how the sensing is exploited for information gathering.

Sensing for Information Gathering

For information gathering, there are two important uses of run-time variables. First, the run-time variables can be used to retrieve information from one source and that information is then used to formulate queries to another source. Second, the run-time variables also can be used to retrieve information which is then used in the selection of the most appropriate information sources. We have already implemented the first use, which is described below, and we investigate the second in (Knoblock & Levy 1994).

The capability for gathering information to use in the formulation of another query can be added to the system by adding two more operators to the domain. The first operator is simply an action to execute a query in the local system and bind the result to "!result". As in UWL, run-time variables are annotated with an exclamation mark. The only precondition of this operator is that the information is available in the local system and the only effect is that the data is bound to the result.

```
(define (operator execute-query)
   :parameters (?query !result)
   :precondition (available local sims ?query)
   :effect (result ?query !result))
```

The second operator, called **use-gathered-info** is the one for retrieving information and using it in the formulation of another query. The heart of this operator is the gather-data precondition, which is a function that determines whether a query can be decomposed such that some of the information can be retrieved and incorporated directly into another query. If so, then it decomposes the original query into a modified query and a sub-query that will get executed first to return an answer. The result will then be inserted directly into the modified query through the run-time binding.

```
(define (operator use-gathered-info)
 :parameters (?source ?host ?query ?mod-query ?sub-query ?answer)
 :precondition (:and (result ?sub-query ?answer)
                     (available ?source ?host ?mod-query)
                (gather-data ?query ?mod-query ?sub-query ?answer))
 :effect (available ?source ?host ?query))
```

Consider the example query described in the previous sections. Instead of executing two parallel queries, the system can first gather the information on the C5 aircraft and incorporate that information directly into the second query. While the two queries must then be done sequentially, it will greatly reduce the amount of intermediate data that needs to be retrieved from the second query. Also, there will be no local processing, so the result can be sent directly to the output. The modified plan is shown in Figure 8. Notice that in this plan the binding of "!result," which is bound to 42 in the bottom part of the plan, is incorporated directly into the query against the **geo** database.

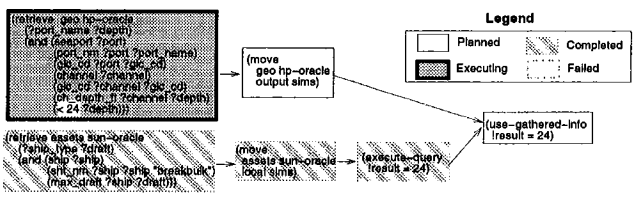

Figure 8: Exploiting Sensing Actions

Related Work

There are a variety of systems that have tightly integrated planning with some combination of execution, sensing, and replanning. There is work on reactive planning (Firby 1987), which emphasizes the ability to react to unexpected situations rather than assume that a plan will usually work. This view is appropriate for some domains, such as robot planning, but not in domains such as information gathering where the cost of execution will usually be much higher than the cost of reasoning about actions. In the classical planning framework, Ambros-Ingerson and Steel (Ambros-Ingerson 1987) developed an integrated planning, execution, and monitoring system called IPEM and introduced the idea of run-time variables for sensing. Owalski and Gini (Olawsky & Gini 1990) focused on the tradeoffs and strategies in choosing when to sense and when to plan. Etzioni et al. developed a language for representing incomplete information (Etzioni *et al.* 1992) and built an integrated system for planning, execution, and sensing called XII (Etzioni, Golden, & Weld 1994) that can represent and reason about locally complete information.

This paper extends this earlier work within the classical planning paradigm with the capability to perform the planning in parallel with the execution. This allows the system to execute actions in parallel, handle new goals as they arrive, replan failed actions while other actions continue to execute, and perform sensing operations simultaneously with other actions. Of the earlier work, only IPEM claims to support execution of actions in parallel, but does not address the issue of resource conflicts and does not consider alternative plans during execution (Ambros-Ingerson 1987, pg.50). Other system have supported planning of simultaneous actions (Currie & Tate 1991, Wilkins 1984), but do not integrate the planning and execution.

The other aspect to this work is the application of the planner to the problem of information gathering. This provides a real-world example of where this planner is useful and extends the state of the art in query processing. Conventional query processors produce a query access plan and then execute it (Jarke & Koch 1984). There is no capability for interleaving the planning and execution, performing sensing operations, replanning due to failures, or handling additional goals. This planner described in this paper provides the capability for a much more flexible and efficient query processor.

Conclusion

This paper presents the planning system, called Sage, which interleaves planning and execution, run continuously and handles new goals as they arrive, performs sensing actions, and recovers from failures that arise, all while continuing to execute actions already in progress. This planner serves as the underlying query planner for the SIMS information mediator (Arens *et al.* 1993, Knoblock, Arens, & Hsu 1994), whose goal is to provide flexible and efficient access to large numbers of information sources. The planning, execution, and sensing are fully implemented. The replanning module currently replans just the failed subgoals and we are in the process of incorporating the ideas from SPA (Hanks & Weld 1992), which will provide a more complete replanning capability. The current system has been used in the domains of logistics planning and trauma care and provides access to data stored in Oracle databases, Mumps databases, and Loom knowledge bases, as well data produced by Lisp programs.

References

Allen, J. F.; Kautz, H. A.; Pelavin, R. N.; and Tenenberg, J. D. 1991. *Reasoning About Plans.* San Mateo: Morgan Kaufmann.

Ambros-Ingerson, J. 1987. *IPEM: Inegrated Planning, Execution, and Monitoring.* Ph.D. Dissertation, Department of Computer Science, University of Essex.

Arens, Y.; Chee, C. Y.; Hsu, C.-N.; and Knoblock, C. A. 1993. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems* 2(2):127–158.

Barrett, A.; Golden, K.; Penberthy, S.; and Weld, D. 1993. Ucpop user's manual (version 2.0). Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington.

Currie, K., and Tate, A. 1991. O-plan: The open planning architecture. *Artificial Intelligence* 52(1):49–86.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning,* 115–125.

Etzioni, O.; Golden, K.; and Weld, D. 1994. Tractable closed-world reasoning with updates. In *Fourth International Conference on Principles of Knowledge Representation and Reasoning.*

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence,* 202–206.

Hanks, S., and Weld, D. S. 1992. The systematic plan adaptor: A formal foundation for case-based planning. Technical Report 92-09-04, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

Jarke, M., and Koch, J. 1984. Query optimization in database systems. *ACM Computing Surveys* 16(2):111–152.

Knoblock, C. A., and Levy, A. 1994. Efficient query processing for information gathering agents. In *Proceedings of the Workshop on Intelligent Information Agents.*

Knoblock, C.; Arens, Y.; and Hsu, C.-N. 1994. Cooperating agents for information retrieval. In *Proceedings of the Second International Conference on Cooperative Information Systems.*

Knoblock, C. A. 1994. Generating parallel execution plans with a partial-order planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.*

MacGregor, R. 1990. The evolving technology of classification-based knowledge representation systems. In Sowa, J., ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann.

Olawsky, D., and Gini, M. 1990. Deferred planning and sensor use. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control,* 166–174.

Penberthy, J. S., and Weld, D. S. 1992. Ucpop: A sound, complete, partial order planner for adl. In *Third International Conference on Principles of Knowledge Representation and Reasoning,* 189–197.

Selinger, P. G.; Astrahan, M.; Chamberlin, D.; Lorie, R.; and Price, T. 1988. Access path selection in a relational database management system. In *Artificial Intelligence and Databases.* Los Altos, CA: Morgan Kaufmann. 511–522.

Wilkins, D. E. 1984. Domain-independent planning: Representation and plan generation. *Artificial Intelligence* 22(3):269–301.