Planning, Executing, Sensing, and Replanning 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, a number of problems arise in gathering information from large networks of distributed information. In this environment, the same information may reside in multiple places, actions can be executed in parallel to exploit distributed resources, new goals come into the system during execution, actions may fail due to problems with remote databases or networks, and sensing may need to be interleaved with planning in order to formulate efficient queries. 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.

1 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 of generating a static plan for processing information and then executing it is inflexible and may be very inefficient if problems arise during query processing. The problem is that there may be many information sources from which to choose, actions may fail, the system has incomplete knowledge about the available information, and new goals may arise at any time.

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

Before describing the integration of planning and execution, we first describe the information gathering task and how it can be cast as a planning problem in a general planning framework (Section 2). Next, we present our approach to tightly integrating planning and execution (Section 3). This integration is used to support planning for new goals, replanning for failure, and the interleaving of sensing actions to gather additional information for planning (Section 4). We compare this work to previous work in planning as well as information gathering and query processing (Section 5). Finally, we conclude with a discussion of the contributions of the paper (Section 6).

2 Planning for Information Gathering

Information gathering requires selecting, integrating, and retrieving data from distributed and heterogeneous information sources in order to satisfy a query. The relevant data must be selected from numerous, possibly overlapping or replicated sources. Integrating the information may be costly, especially when combining data from different sites. Retrieving the information may be time consuming due to the distribution of data and the contention for limited resources.

To solve this problem, we have developed a planner

*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 Number F30602-91-C-0081, and in part by the National Science Foundation under Grant Number IRI-9313993. The views and conclusions contained in this paper are those of the author 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.

called Sage that builds on the UCPOP partial-order planner [Barrett *et al.*, 1993]. 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. We extended this planner to support simultaneous action execution and to tightly integrate planning and execution. The execution is presented in the next section, and the support for simultaneous actions was previously addressed in [Knoblock, 1994] and will be briefly described here.

Partial-order planners, such as UCPOP, produce plans with actions that are unordered. However, if two actions are left unordered they can be executed in either order, but not simultaneously. To execute actions in parallel in a partial-order planner requires that (1) actions can be executed simultaneously without changing the outcome of the individual actions, and (2) any potential resource conflicts must be captured in the representation of the operators in order to avoid conflicts during execution. We assume that the first condition holds (as it does in the information gathering domain described below) and we extended the planner to support the second condition. To support reasoning about resources, we added an explicit resource declaration to the action language, which describes the resources required when executing an action. We also augmented the planner to identify and remove potential resource conflicts. With these extensions, any actions left unordered in the final plan can be executed simultaneously.

In the remainder of this section we describe how the information gathering task is cast as a planning problem in Sage. This problem requires producing a plan for generating a requested set of data. This involves selecting the sources for the data, the operations for processing the data, the sites where the operations will be performed, and the order in which to perform the operations. Since data can be retrieved from multiple sources and the operations can be performed in a variety of orders, the space of possible plans is large.

An information gathering goal consists of a description of a set of desired data as well as the location where that data is to be sent. For example, Table 1 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 *et al.*, 1994]. The goal also 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.

The initial state of a problem defines the available information sources (e.g., databases) and the servers (e.g., an Oracle DBMS) they are running on. The example shown in Table 2 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, a description of the contents

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

Table 1: An information gathering goal

of the information sources 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.

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

Table 2: An initial state

For this domain, Sage uses a set of ten general operators to plan out the processing of a query. They 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 a **select-source** operator for selecting the information source for retrieving a set of data. The other operators perform additional processing of data (**select**, **compute**, and **assignment**) or reformulate queries using background knowledge (**generalize**, **specialize**, **definition**, and **decompose**). Each operator 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 Table 3, which defines a join performed in the local system. This operator is used to achieve the goal of making some information available in the local knowledge base of the SIMS information mediator. It does this by partitioning the request into two subsets of the requested data, retrieving that information into the local system, and then joining the 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 function that produces the relevant partitions of the requested data.

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

Table 3: The join operator

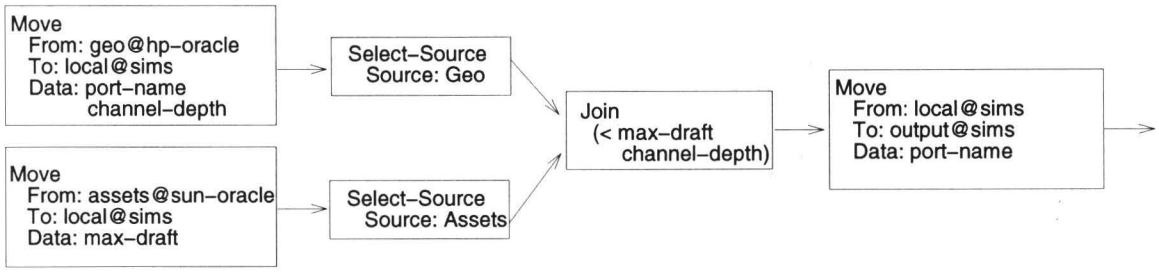

Figure 1: An information gathering plan

This planning domain differs from many of the domains that previous planning work has focused on in two significant ways. First, there are few interactions between the operators. The main source of interaction arises in handling resource conflicts when two operators require access to the same server. Second, it is not sufficient to find any solution to a problem; the goal is to find an efficient solution. The first difference makes the problem somewhat easier, while the second difference makes the problem significantly harder since it may require searching a large space of plans.

In order to generate query access plans efficiently, we have carefully constrained the space of possible plans. We wrote the operators such that they generate only the relevant portions of the search space. Some examples of this are: first, the operators only reason about joins in the local system, since joins in the remote systems will be handled by the remote database management system and the planner has no control over how or in what order these are performed. Second, the operators consider only joins across data that are distributed in different information sources. It will generally be less efficient to pull two sets of information from the same information source and perform the join locally rather than in the remote source. Third, since we usually do not have write access to the remote databases, information can only be moved from the remote systems to the local system or directly to the output. However, even with a set of carefully designed operators, the search space may still be very large since the operations can be performed in different orders, and there may be multiple replicated and overlapping sources from which the information can be retrieved.

To further constrain the overall search for an efficient plan, we also employ standard database estimation techniques to write an evaluation function to guide the search. The planner uses the evaluation function in a branch-and-bound search, estimating the cost of each intermediate plan and selecting the plan with the lowest overall execution cost. The cost of each operation is estimated by maintaining information about the size of each relation and the number of different possible values for each attribute of a relation. Assuming a uniform distribution of the data, we then estimate the amount of intermediate data that will be retrieved and manipulated, which is usually the dominant cost in handling multidatabase queries. Using the estimated cost of each operation, we can then compute an estimate for the overall cost of a plan, taking into account the parallelism of some of the actions. The evaluation function allows the planner to compare different partial plans; those plans that are more expensive than the plan eventually selected will never be expanded further.

The final plan generated for the example query in Table 1 is shown in Figure 1. This plan shows where the information is retrieved from and how the information is manipulated to produce the requested data. The system works backward from the goal to produce a plan to retrieve the data. In this particular plan the final **move** operator is used to achieve the original goal of sending the requested data to the output; it also generates the subgoal of getting the data into the local system. Next, the system considers how to get the data into the local system and since the information is not available in any single information source, it selects the **join** operator, which decomposes the original goal into two simpler information goals. Each of these simpler goals is then achieved by using the **select-source** operator to select a relevant source for each of the requests and translate the requests into subgoals that use the terminology of the selected information source. These goals are in turn achieved by moving the information from the remote information sources into the local system. When this plan is executed, all of the information is brought into the local SIMS mediator, where the draft of the ship can be compared against the depth of the seaports. Once the final set of data has been generated, it is sent to the output.

The approach of searching the space of plans to find the best one 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. An important difference from traditional query planning systems is that in those systems the source from which the information is to be retrieved is fixed, whereas part of the planning process described here includes the selection of an appropriate information source. While this makes the problem harder, it also provides a much more flexible approach to integrating distributed and heterogeneous sources of information.

So far we have described the approach to generating query plans for information gathering in a distributed and heterogeneous environment. In addition to gener-

ating a plan, the system must also execute it. However, unlike traditional database environments, there are a number of problems and issues that arise when dealing with distributed and autonomous information sources. Information sources may be unavailable, queries may fail, new information requests may arise that compete for resources with the currently executing plan, and additional information may be required to select an appropriate plan or formulate an efficient query. In the remainder of this paper we will describe how planning and execution are tightly integrated and how this integration is used to address the issues that arise during execution.

3 Integrating Planning and Execution

Planning and execution are tightly integrated by considering 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 is required for a complete plan, executing each of the actions is also part of the final result.

Sage keeps track of the current status of every action in the plan by marking them as either *unexecuted*, *executing*, *completed*, or *failed*. This is similar to how execution was integrated into IPEM [Ambros-Ingerson, 1987]. The underlying planner, UCPOP, maintains a list of *flaws*, which is an agenda of things that need to be done to complete a particular plan. These flaws include *open conditions*, which are subgoals 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 execution in Sage by adding two new types of flaws: an *unexecuted* action flaw and an *executing* action flaw. Whenever a new operator is added to a plan, the corresponding flaw indicating that the action is unexecuted is also added to the agenda. The *executing* flaw is used to handle the fact that actions are not instantaneous and in some cases may take considerable time. A plan is not complete until all *unexecuted* and *executing* flaws have been removed.

The choice of when to execute an action in a plan is important, since undoing an executed action may be costly or impossible. An action cannot be executed until every precondition of the action has been both planned and achieved by executing the preceding actions. Even after an action is executable, Sage delays execution as long as possible to avoid committing to a partially constructed plan prematurely. Once an action has been executed, it is viewed as a commitment to the plan in which the action occurs – the planner cannot consider any plans that are not refinements of the plan being executed. The idea is that the planner should find the best complete plan before any action is executed. Then once execution is initiated, it resolves any failed subplans or new goals before executing the next action. This means that the planner will never execute an action until the corresponding plan is selected as the best available.

Since executing an action may take considerable time, the planner cannot simply execute an action and wait for the results. Instead, Sage creates a subprocess that executes the action and notifies the planner once it has completed. In order to keep track of the actions currently being executed, the corresponding *unexecuted* flaw is removed from the agenda and the *executing* flaw is added. At any one time there may be a number of actions that are all executing simultaneously. On each cycle of the planner, the system checks if any executing actions have completed. Once an action is completed, the *executing* flaw is removed from the agenda. If it completes successfully, the action is marked as completed. Other actions that depend on this action may now be executable if all of the other preceding actions have also been executed. If an action fails, the failed portion of the plan is removed and then replanned, as described in the next section.

Sage's top-level algorithm for tightly integrating planning and execution is summarized in Table 4. The planner starts with an initial plan, where the goals are the open conditions. Initially, the set of *current plans* contains only this initial plan. It repeats the algorithm until it produces a plan in which every action has been executed. The planner considers only refinements of the *current plans*. Whenever an action is executed, an action terminates, or a new goal is added, the set of *current plans* is replaced by a new set containing only this new plan. The first two conditions in this algorithm ensure that the planner finds a plan with no open conditions or threats before it commits to a plan and initiates any actions.

Remove a plan from the set of *current plans* and apply the first applicable condition:

- If there are any threats, resolve them by adding additional constraints to the plan. Add the possible refinements to the *current plans*.
- If there are any open conditions, add additional actions or ordering links to achieve them. Add the possible refinements to the *current plans*. (As described in the next section, open conditions that contain runtime variables for sensing will be postponed.)
- If any executing actions have completed:
  - If the action completed successfully, record the results and update the plan. If the plan is complete, return the results. Otherwise, replace the *current plans* with this new plan.
  - If the action failed, remove the failed portion of the plan, update the model to avoid generating the same plan again, and replace the *current plans* with this new plan.
- If there are any new goals to solve, add them to the open conditions and replace the *current plans* with this new plan.
- If any unexecuted actions are now executable, create a process to execute them and replace the *current plans* with this new plan.

Table 4: Algorithm for planning and execution

This algorithm supports simultaneous planning and execution. Before the system initiates execution of any

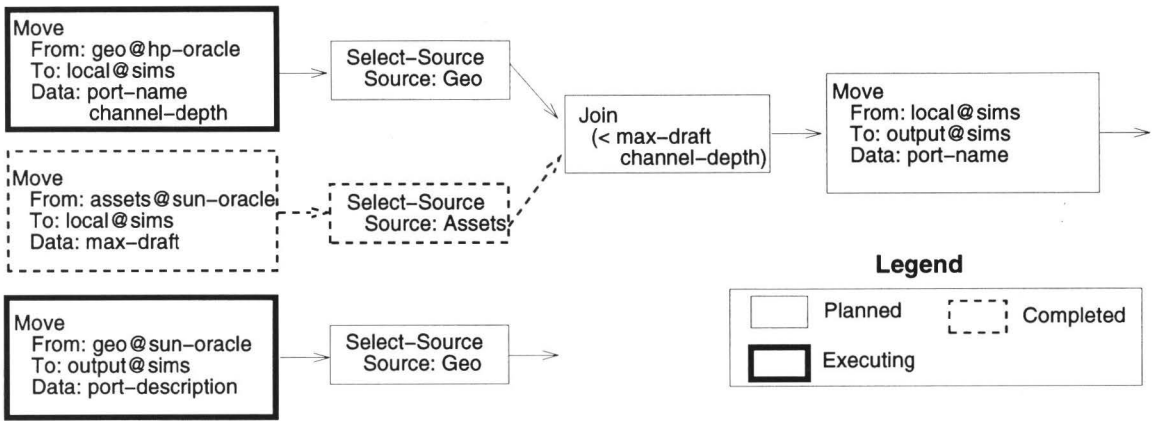

Figure 2: Planning for new goals

action, it constructs an initially complete plan. However, once execution starts, an action could fail, a new goal could arise, or the system may require additional information (sensing) to continue planning. In any of these cases, once the new open condition has been added to the list of flaws, the system can augment the executing plan to achieve these conditions while it continues executing any actions that have already been initiated. In the next section we describe these capabilities in more detail.

4 Advantages of Integrating Planning and Execution

Integrating the planning and execution allows the system to plan for new goals as they arrive, replan failed actions, and exploit sensing operations, all while the system is executing other actions in a plan.

4.1 Planning for New Goals

Interleaving planning and execution allows the system to handle new goals while the system is in the midst of executing a plan that achieves some other goals. This is important, since execution may require substantial amounts of time and it may be impractical and inefficient to wait for one task to complete before starting the next task. In addition, it may not be possible to treat the new goal as an independent task since it may compete with the executing plan for the same resources. The handling of new goals is captured in the algorithm described in Table 4. When a new goal arises, the system adds this goal to the currently executing plan and then refines that plan to solve the goal.

Consider an example where a new goal is given to the system while it is executing the plan in Figure 1. Assume that the system has already executed some of the actions and is in the midst of executing others, as shown in Figure 2. When a new goal arises to retrieve the description of the Long Beach seaport, the planner notices the pending goal on the next cycle and then searches for 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) continues to execute, since actions are run as separate processes.

The resulting plan is shown in Figure 2. 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.

4.2 Replanning Failed Actions

Integrating planning and execution allows the system to gracefully handle action failures and replanning. Since the planner may have expended considerable effort in executing a plan so far, we want to avoid throwing out the entire plan and starting from scratch when an action fails. Instead, the planner should replan the failed portion of the plan, while maintaining as much of the executing plan as possible. This is currently supported by requiring the designer of a domain to define a set of domain-specific failure handlers. When a failure occurs, the failure handler is called with the action that failed and the type of failure, and the failure handler is expected to remove the failed portion of the plan and update the model to avoid the same failure when the failed actions are replanned. This replanning can be performed while other unaffected actions continue to execute. A more complete replanning capability could be incorporated by using the approach developed in the Systematic Plan Adaptor (SPA) [Hanks and Weld, 1992], which systematically searches the space of plan modifications.

In the information gathering domain, the ability to replan upon failure can be exploited to handle query failures by redirecting a query to a different informa-

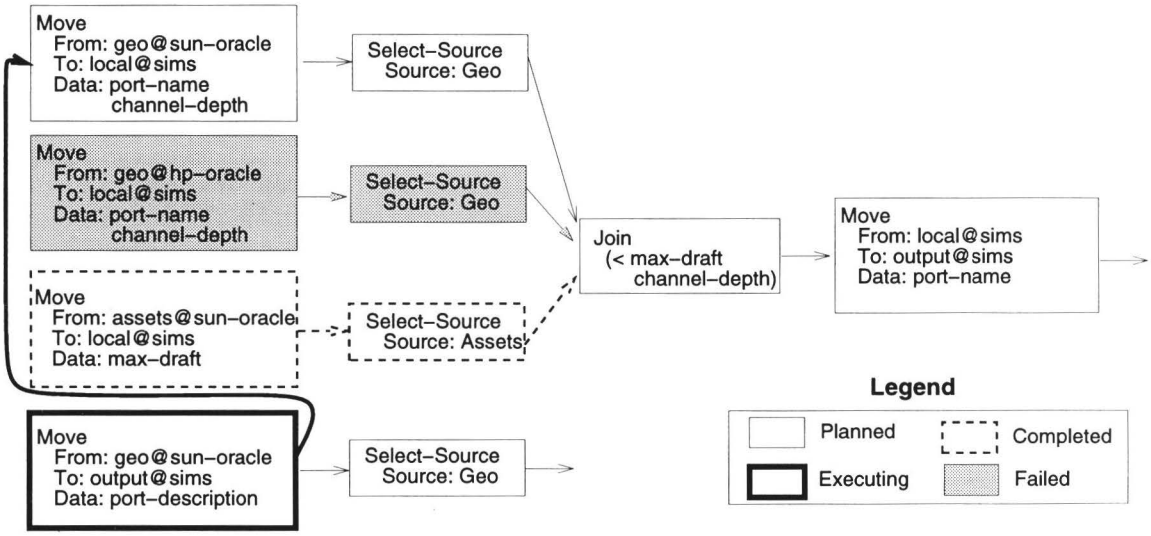

Figure 3: Replanning a failed plan

tion source. An execution failure may occur because a database or network is down. In this case the failure handler would remove the actions for retrieving the data from a specific information source and would mark the information source as unavailable to avoid generating the same plan. The planner would then attempt to replan the query; if another information source is available it would generate an alternative plan.

An example of a failed action that can be replanned is shown in Figure 3. The actions in the shaded 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. This constraint prevents the replanned move action from being executed until this other action completes.

4.3 Sensing to Plan

Integrating planning and execution allows the system to interleave sensing actions with the planning. Earlier work on sensing in planning [Ambros-Ingerson, 1987; Etzioni *et al.*, 1992] proposed the idea of incorporating run-time variables in the planner to allow the planner to reason about the sensed information. 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. Sage supports run-time variables and delays working on any open condition that involves such a variable. However, unlike previous planners, Sage can begin execution of other actions while it is waiting for the sensed information and then continue planning while these actions continue to execute.

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 and Levy, 1995].

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, shown in Table 5. 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. Note that the system will have to generate a subplan and execute it in order to get the information into the local system.

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

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

Table 5: Operators for sensing

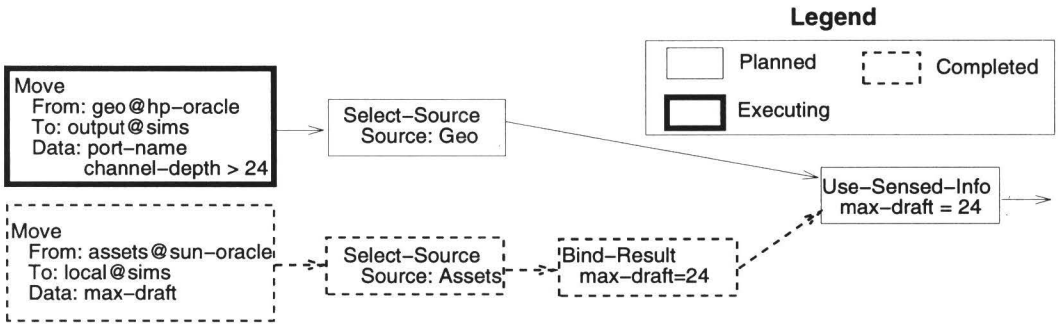

Figure 4: Exploiting sensing actions

The second operator, called **use-sensed-info**, retrieves information and uses 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, which will get executed first to return an answer. The result will then be inserted directly into the modified query through the run-time binding.

Consider the example query described in the previous sections. Instead of executing two parallel queries, the system can first gather the information on the ship draft and incorporate that information directly into the second query, as shown in Figure 4. In this plan the binding for the "max-draft," is incorporated directly into the query against the **geo** database. 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.

5 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 (e.g., [Firby, 1987; Beetz and McDermott, 1992]), 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 a partial-order planning framework, Ambros-Ingerson [1987] developed an integrated planning, execution, and monitoring system called IPEM and introduced the idea of run-time variables for sensing. Olawsky and Gini [1990] focused on the tradeoffs and strategies in choosing when to sense and when to plan. Etzioni et al.[1992] developed a language for representing incomplete information and Etzioni et al.[1994] built an integrated system for planning, execution, and sensing called XII that can represent and reason about locally complete information. We have built on many of the ideas from the earlier work within the partial-order planning paradigm and extended them to support simultaneous planning and execution and build an integrated system for information gathering.

The other aspect to this work is the application of the planner to the problem of information gathering. The XII planner [Etzioni *et al.*, 1994], which is used in the Unix Softbot [Etzioni and Weld, 1994], also supports execution and sensing for information gathering. Compared to Sage, the Softbot reasons about the information at a different level of granularity. Instead of representing general actions for manipulating data, each operator corresponds to a Unix command. The advantage of their approach is that it provides finer-grained control and reasoning of the information. The disadvantage is that it would be impractical to efficiently reason about and manipulate large amounts of information. Information gathering is also similar to conventional query processing in databases. These systems generate a query access plan and then execute it [Jarke and Koch, 1984]. There is no choice of which information source is used and no capability for interleaving the planning and execution, performing sensing operations, replanning due to failures, or handling additional goals.

6 Discussion

This paper presented a planning system, called Sage, which tightly integrates planning and execution, runs 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. The contributions of this work are twofold. First, we extended the previous work by tightly integrating these components and adding the capability to execute actions simultaneously with the planning, replanning, and sensing. Second, we demonstrated that the resulting planner can be effectively applied to the problem of information gathering from distributed and heterogeneous information sources.

In this work we started with a real-world planning application and identified the issues that had to be addressed to solve this problem. While there is a significant amount of previous work on planning that we could build on, the emphasis and assumptions in previous work do not closely match the problems that arise in this domain. For example, in terms of generating plans, the interactions between actions do arise, but they are not the dom-

inant problem. Issues that are important in this domain are finding high quality plans, exploiting parallelism in the plans, and planning and executing simultaneously to support planning for new goals, replanning and sensing. In order to put all of this work together and turn it into a practical planning system, the resulting planner makes some simplifying assumptions that may not hold in other domains. However, the basic architecture is quite general and has been demonstrated in a real-world application.

Sage serves as the underlying query planner for the SIMS information mediator [Arens *et al.*, 1993; Knoblock *et al.*, 1994]. The goal of SIMS is to provide flexible and efficient access to large numbers of information sources. We have implemented the planning, execution, replanning, and sensing as described in this paper. The current system has been used in the domains of logistics planning and trauma care and provides access to data stored in a variety of systems that are distributed at various sites.

Acknowledgments

Thanks to the other members of the SIMS project, Yigal Arens, Wei-Min Shen, Chin Chee, Chun-Nan Hsu, Jose-Luis Ambite, and Sheila Tejada, for their work on developing the rest of the system. Also, thanks to Yolanda Gil, Kevin Knight, Qiang Yang, Sheila Coyazo, and the anonymous reviewers for their comments on earlier drafts of this paper.

References

[Ambros-Ingerson, 1987] Jose Ambros-Ingerson. *IPEM: Integrated Planning, Execution, and Monitoring.* PhD thesis, Department of Computer Science, University of Essex, 1987.

[Arens *et al.*, 1993] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[Barrett *et al.*, 1993] Anthony Barrett, Keith Golden, Scott Penberthy, and Daniel Weld. UCPOP user's manual (version 2.0). Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington, 1993.

[Beetz and McDermott, 1992] Michael Beetz and Drew McDermott. Declarative goals in reactive plans. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, pages 3–12, College Park, MD, 1992.

[Etzioni and Weld, 1994] Oren Etzioni and Daniel S. Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37(7), 1994.

[Etzioni *et al.*, 1992] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 115–125, Cambridge, MA, 1992.

[Etzioni *et al.*, 1994] Oren Etzioni, Keith Golden, and Dan Weld. Tractable closed-world reasoning with updates. In *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany, 1994.

[Firby, 1987] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, 1987.

[Hanks and Weld, 1992] Steven Hanks and Daniel S. Weld. 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, 1992.

[Jarke and Koch, 1984] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[Knoblock and Levy, 1995] Craig A. Knoblock and Alon Levy. Exploiting run-time information for efficient processing of queries. In *Working Notes of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*, Palo Alto, CA, 1995.

[Knoblock *et al.*, 1994] Craig Knoblock, Yigal Arens, and Chun-Nan Hsu. Cooperating agents for information retrieval. In *Proceedings of the Second International Conference on Cooperative Information Systems*, Toronto, Canada, 1994.

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

[MacGregor, 1990] Robert MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann, 1990.

[Olawsky and Gini, 1990] Duane Olawsky and Maria Gini. Deferred planning and sensor use. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 166–174, San Diego, CA, 1990.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 189–197, Cambridge, MA, 1992.

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

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