Unified Information and Control Flow in Hierarchical Task Networks

Mike Williamson and Keith Decker and Katia Sycara
The Robotics Institute, Carnegie-Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
(mikew,decker,sycara)@cs.cmu.edu

March 20, 1996

Abstract

Much recent planning research has focused on two related issues. First, there has been a strong interest in information-gathering (or "sensing", or "knowledge-producing") actions. Second, has been an investigation of plans with sophisticated control structures, such as conditional branches and loops. But the combination of these two lines of research poses a representational problem: plans with information-gathering actions that can be executed more than once can have complex information-flow and control-flow relationships. In this paper, we present a framework for the representation and execution of hierarchical plans with information producing actions, conditional branches, periodic actions, and loops. Our framework subsumes several techniques found in the recent literature.

1 Introduction

What is a plan? The answer to this question would seem to be a fundamental characteristic of any planning formalism. Historically, there has been a good deal of research aimed at developing representations for actions, but this work has generally been done in the context of planning formalisms that adopt a very limited definition of plans, namely that a plan is a (partially-ordered) *sequence* of primitive actions. This definition of plans has been widely accepted in both the generative and the hierarchical task network (HTN) planning paradigms. Recently, though, there has been a strong interest in plan representations that support sophisticated control flow, such as parallel execution [9], conditional branching [15, 3] and loops [17, 11, 14, 8]. These developments have gone hand-in-hand with the creation of models for *informative* (a.k.a. "sensing", "information-gathering") actions [12, 5]. The two developments are closely interrelated, since contingencies in a plan are only meaningful if new information becomes available, and conversely, sensing the world is most useful when doing so can have some impact on one's course of action.

1. Retrieve the initial contents of the web page.
2. Periodically retrieve the current contents of the page.
3. Compare the results of actions (1) and (2).
4. If action (3) indicates a difference, give notification of the new content, and signal successful completion of the task.

Figure 1: An informal plan to monitor a web page for change.

This paper will discuss two additional kinds of control flow in plans: 1) *periodic actions*, which are performed repeatedly at specific intervals, and 2) *triggered actions*, which are performed (perhaps repeatedly) in response to external events. There is a clear motivation for allowing such actions: they provide the only reasonable way to perform many useful tasks. For example, consider the task of monitoring a web page for new information. There is a simple plan for this task consisting of four actions, shown in Figure 1. This example illustrates the control-flow and information-flow issues that arise when a plan contains repetitive, information-producing actions. How does the execution system determine when to execute actions (3) and (4)? How does action (3) obtain the information generated by actions (1) and (2)?

Existing planning formalisms explicitly describe control flow in terms of *ordering relationships* between actions, (i.e. $A \prec B$ denoting that action A must be performed before action B). But these relationships are insufficient to distinguish between a number of distinct control relationships that might arise in plans with repetitive actions. In the plan in Figure 1, for example, both actions (1) and (2) must precede action (3) the *first* time each is executed. But thereafter, action (3) stands in very different control relationships to actions (1) and (2); it must be performed again when and only when action (2) has been repeated, whereas action (1) need *not* be repeated in order to reactivate action (3). In order to represent plans such as the one in Figure 1 we need a formalism capable distinguishing these control-flow relationships.

We hold the position that most control flow relationships in a plan are *derivative* from information flow relationships. This paper will present an integrated representation for information and control flow in hierarchical task networks. Our perspective is essentially pragmatic; the formalism we present arose from our need to represent and execute hierarchical task networks in our agent architecture. We have implemented this formalism in an Internet-based multi-agent information system, WARREN, which uses cooperative, autonomous agents to support financial portfolio management.

1.1 The agent architecture

A brief sketch of our agent architecture will assist in understanding the issues we are addressing. The internal structure of our agents is similar to the DECAF architecture [2]. An agent comprises concurrent planning, scheduling, and execution processes operating on a

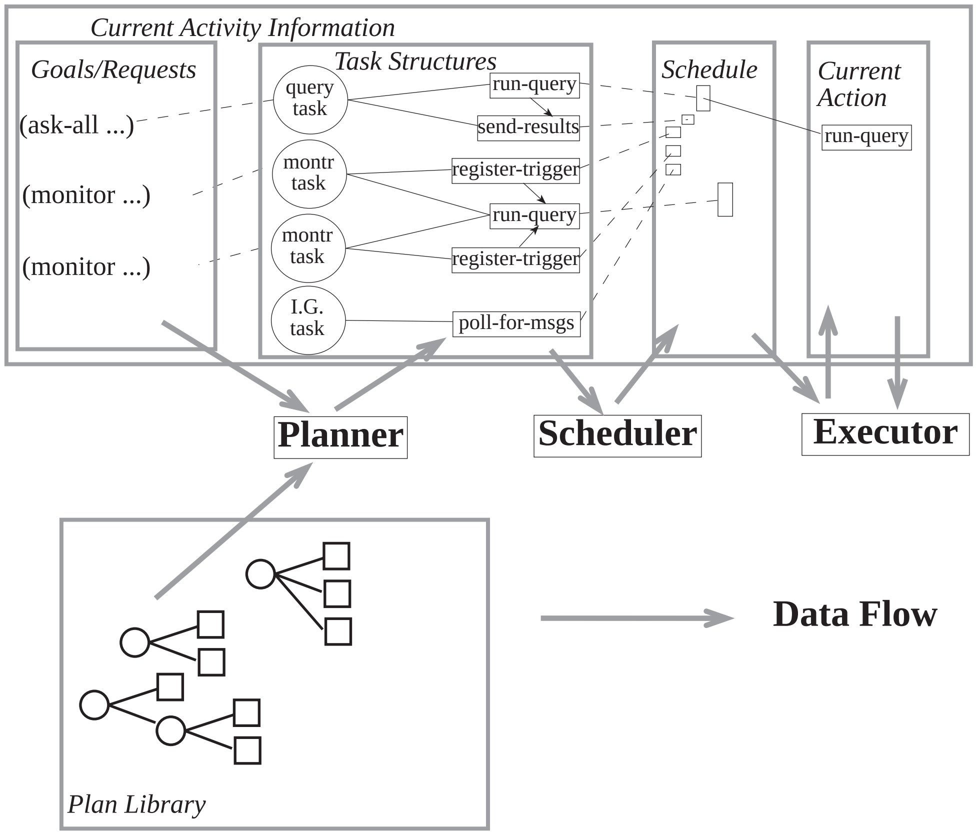

Figure 2: An overview of the agent architecture.

shared task agenda (Figure 2). The planner evaluates the agent's objectives, formulates top-level tasks (goals), and elaborates hierarchical task networks for each top-level task. The task networks consist of higher-level tasks (which must be reduced by the planner to some network of subtasks), and primitive tasks (actions) which may be executed directly. The scheduler's role is to determine when each action should be executed. That is, the scheduler is responsible for making control-flow decisions about the plan. The executor actually carries out an action (in our case by directly invoking a code object attached to the action). Action execution can have external effects on the state of the world, and internal effects on an agents state of knowledge. If an action produces information, that information may need to be routed to other actions that utilize the information. Thus, the executor is responsible for carrying out information-flow within the plan.

Caveat We are designing and building an architecture for distributed, cooperative software agents. Our representation is strongly motivated by the kinds of tasks and actions that arise in such an agent, particularly information-gathering actions which utilize Internet resources, and communicative actions with other agents. The applicability of our representation to other kinds of agents in other domains may be limited.

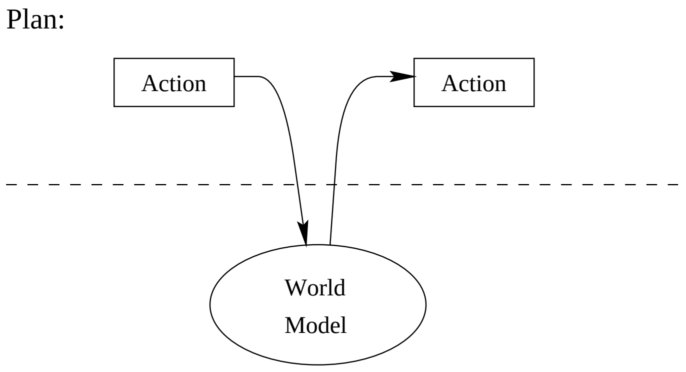

Figure 3: Indirect information flow through an agent's world model.

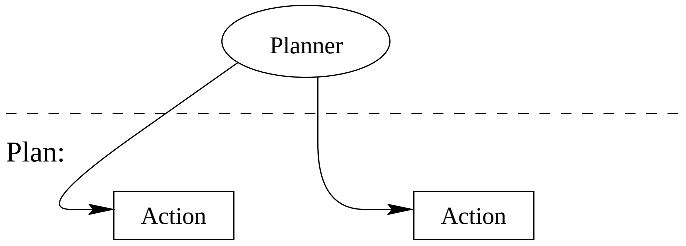

Figure 4: Information flow from the planner by parameter binding.

2 Background

Information flow within plans has received little explicit study. Regarding the formal representation of actions themselves, an action's information requirements have sometimes been framed in terms of preconditions on the state of an agent's knowledge, while the information produced by an action is similarly described in terms of its effects on that knowledge [12, 13]. Thus, information flow in a plan is indirect, taking place through an agent's corpus of beliefs about the world. (See Figure 3.) This work has been primarily theoretical in nature, and does not address practical issues that arise during the execution of plans containing such actions.

The most ubiquitous mechanism for information flow is *parameter binding*. Most planning systems employ some sort of schematic action representations, where the preconditions (and effects) may make reference to variable parameters which are bound by the planner when the plan is created. This allows information to be statically provided from outside the plan before execution begins (Figure 4. A Blocksworld `Stack` action, for example, needs to be informed of *which* block to stack, and *where* to stack it; this information is provided by the planner binding the action's variable parameters to specific values.

Several planning formalisms [1, 5, 10] extend the concept of variable parameters to include "runtime variables." Runtime variables appearing in the effects of an action are bound to some particular value when that action is executed, and may be used in the preconditions of subsequent actions. Thus, runtime variables are an explicit representation of the information flow relationship between producers and consumers (Figure 5). UWL [5] also uses runtime variables to control the flow of execution within a plan. At any point, a plan may bifurcate

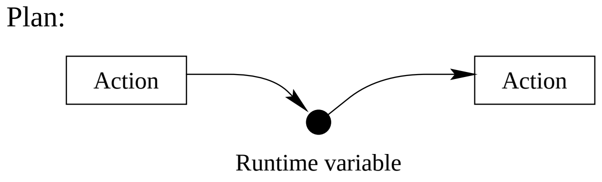

Figure 5: Information flow by binding runtime variables.

into two branches depending on whether or not a runtime variable takes on a particular value.

The conditional planners CNLP [15] and C-BURIDAN [3] use a somewhat different method to control execution flow based on gathered information. In these systems, the execution of an action leads to one of a number of prespecified *outcomes* ("consequences" in C-BURIDAN's parlance). The outcomes of an action are partitioned into discernible equivalence classes representing the distinctions between outcomes that are apparent to the agent at plan-execution time.[1] Each equivalence class is annotated with an *outcome label.* The information produced by the execution of a plan consists of the sequence of outcome labels produced by the execution of each action. Subsequent actions in the plan may be annotated with a *context*, a conjunction of outcome labels. An action is executed only if its context is consistent with the outcome labels produced by all previously executed actions. The outcome/context mechanism thus provides an explicit representation of the control-flow relationships in a plan. Reactive task network architectures, such as TCA [16] and RAPS [7], provide similar models of control flow

Runtime variables and outcomes/contexts are sufficient to capture the information-flow and control-flow relationships in a conditional sequential plan, where each action is executed at most once. Smith and Williamson [17] extended the outcome/context mechanism to support control flow in plans with loops, but did not address information flow. In this paper we present a unification and extension of all these techniques, which supports the representation and efficient execution of plans with periodic actions, externally triggered actions, and loops.

3 Information-producing and consuming actions

In this section, we describe our representation for information flow within plans. The foundation of this representation is a description of the information requirements of actions, and of the information-producing abilities of actions. We will also discuss how information flow is used to control plan execution.[2]

[1]In CNLP all outcomes are distinguishable, so each is in its own discernible equivalence class.

[2]We discuss here only those aspects of our representation that relate to information flow, and omit the more traditional aspects of action modeling such descriptions of the actions preconditions and effects. We believe that our representation for information flow could be used with a variety of different action representations, from simple, propositional, deterministic representations such as STRIPS to the more expressive representations supporting conditionality, quantification, non-determinism and/or metric attributes.

3.1 Consuming information: provisions

The information needs of an action are represented by a set of *provisions*. Provisions can be thought of as a generalization of both parameters and runtime variables. (In fact, we define an action's *parameters* to be a subset of its provisions that obey certain conditions, described below). Like parameters and runtime variables, each provision has a symbolic name. For example, an action to fetch a page from the web might have a provision named `URL`. Unlike runtime variables (and outcome labels), the scope of provision names is local to each action, so there is no necessary connection between different actions with the same provision name.

The primary difference between provisions and parameters or runtime variables is that instead of being bound to a single value, each provision has an associated *queue* of values. When information is supplied to some provision of some action, it is inserted into the queue. Information may be supplied statically by the planner when a plan is being composed, or dynamically during plan execution, either as the result of the execution of some other action or because of the occurrence of some external event (such as the arrival of a message to the agent).

Individual action instances may be designated by the planner as *periodic* or *aperiodic*. An aperiodic action will only be scheduled for execution once. A periodic action is rescheduled upon execution for subsequent re-execution according to its associated period. An action is *enabled*, and thus eligible for execution, when there is at least one value queued for each of its provisions. Upon execution, the action "consumes" its provisions by removing them from the queue. If multiple values are available on a provision's queue, the action may consume all values, thus disabling itself until new information arrives, or consume a single value, thus leaving itself enabled and eligible for future re-execution. An action that has no provisions is always enabled; if it is periodic it will be perpetually available for re-execution.

Parameters are a subset of the provisions with somewhat different behavior. First, a parameter may only be provided once, so its queue will either contain no values or a single value. Second, the value of a parameter is *not* consumed during execution, so its value will remain available for future executions of the action. Note that the distinction between parameters and provisions in our system is not the same as the distinction between plan-time and runtime variables in previous planners; our parameters may have their values supplied dynamically during plan execution, while provisions might have values supplied ahead of time by the planner.

3.2 Producing information: outcomes and results

Where does the information come from that is supplied to the provisions of actions? As we said above, it may be provided by the planner, and it may be produced by the occurrence of external events (more on both of these later), but the most common source of information is the execution of actions.

In our framework, the execution of an action produces an *outcome* and a *result*. The outcome is one of a finite set of predesignated symbols (equivalent to the outcomes of CNLP and the observation labels of C-BURIDAN). The result can be any arbitrary piece of information returned by the code object that implements the action. An example will help make clear the intended difference. Consider the action to retrieve a web page. The outcomes of such an action might be `OK` and `ERROR`, depending on whether the action succeeds in fetching the page, or fails for some reason. If the outcome is `OK`, the result of the action would be the web page itself. In the event of an `ERROR` outcome, the result might be some description of the error.

Having both outcomes and results may seem redundant, but there is a good practical motivation. The outcomes represent a partitioning of the (perhaps infinite) set of possible results into equivalence classes which are distinguished by their expected use in routing information. Outcomes allow the executor to efficiently route results (as described below) without needing to actually examine those results. Thus, outcomes allow control flow to be decoupled from the specific content of information produced.

4 Information flow relationships

The previous section described how individual actions produce and consume information. A *task network* consists of a set of actions and a set of information-flow relationships between those actions. In this section, we will describe how these information-flow relationships are represented in our system. We will first discuss those relationships which may occur between actions in a non-hierarchical (or "flat") task network, then later address the additional relationships that arise in the hierarchical case.

4.1 Provision links

Information flow relationships between actions in a non-hierarchical task network are represented by *provision links*. Each provision link is a tuple $\langle P, \omega, C, \pi \rangle$ where P and C are actions, ω is an outcome of P, and π is a provision of C. The meaning of such a link is that if the execution of P gives rise to outcome ω, its result is supplied to provision π of action C.

We will demonstrate the use of provision links with an example. Recall the monitoring plan shown in Figure 1. Figure 6 shows the task network representation of that plan in our framework. Periodic actions are represented by large boxes with rounded corners, while aperiodic actions are represented by large, square-cornered boxes. The small boxes on the left side of each action represent parameters (with square corners) and provisions (with rounded corners).

The two fetch actions will both always be enabled, since their single parameter is provided by the planner. The first one will only be executed once because it is not periodic, but the second one will be executed repeatedly. (It stays enabled since parameters are not consumed on execution.) The comparison action will be enabled when both of the fetches have supplied results. Upon execution, the value of PAGE2 (a provision) will be consumed, so the action will be disabled. It will be re-enabled when the periodic fetch action is executed again, and a new value is supplied. The final notification action will be enabled whenever the comparison has a SAME outcome.

Note that provision links obviate the need for the ordering constraints that are used in many existing planning formalisms. The planner instead describes the information-flow relationships that hold between actions. A correct ordering of actions is determined by the

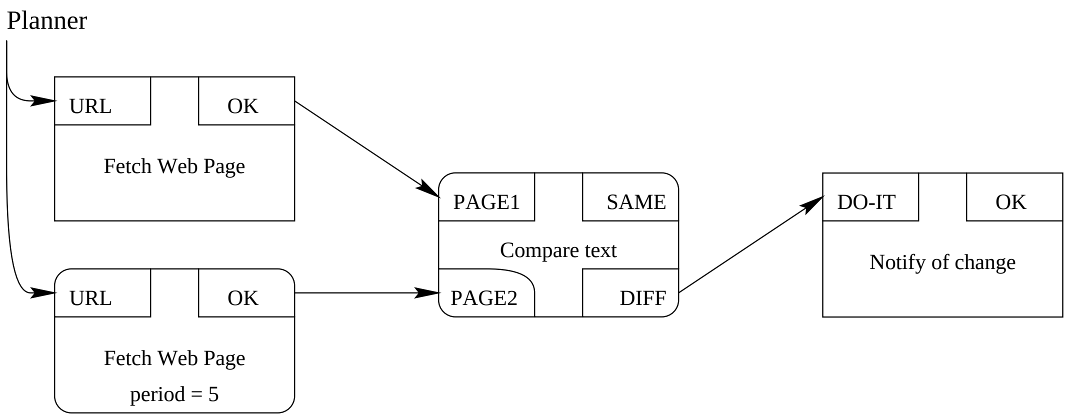

Figure 6: A task network for a simple monitoring plan.

scheduler, which selects an action for execution if and only if all the required information is available.

4.2 Inheritance and disinheritance

In our hierarchical task network planning framework, a task is *reduced* by instantiating a set of subtasks. Unlike some recent formalizations of HTN planning [4], we do not replace the reduced task with its subtasks, but instead represent the task network as a tree-like structure. The definition of a reduction specifies how the provisions and outcomes of the new subtasks relate to each other, and to their parent task. In order that our reduction schemas be modular, the provision links described in the previous section can only be used between *siblings* (i.e. immediate subtasks of a common higher level task). The provisions and outcomes of the subtasks are related to their parent task by two other kinds of relationships: *provision inheritance* and *outcome disinheritance*.

A provision inheritance link is a tuple $\langle T, \pi_T, S, \pi_S \rangle$, where S is a subtask of T, π_T is a provision of T, and π_S is a provision of S. The meaning of such a link is that any value supplied to π_T will be passed on to π_S.

Similarly, an outcome disinheritance link is a tuple $\langle S, \omega_S, T, \omega_T \rangle$, where S is a subtask of T, ω_S is an outcome of S, and ω_T is an outcome of T. Such a link indicates that if the execution of S results in outcome ω_S, the supertask T will have the outcome ω_T. Additionally, the *result* of S will also be passed on as the result of T.

We will illustrate these relationships with a simple example. The agents in our system, WARREN, often gather information by communicating with other agents. An abstract task of answering a particular query could be broken down into two subtasks as shown in Figure 7. The first subtask determines which agent should be used to answer the query, and the second subtask asks that agent to answer the query. Both subtasks inherit the query from their parent. The Determine-Agent subtask has two possible outcomes. If an appropriate agent is known, its name is provided to the Ask-Agent-Query subtask. Otherwise, the failure outcome is propagated upwards, and indicates the failure of the parent task. The task that actually answers the query will be enabled when it is provided an agent's name. The successful completion of this subtask defines the successful completion of the parent.

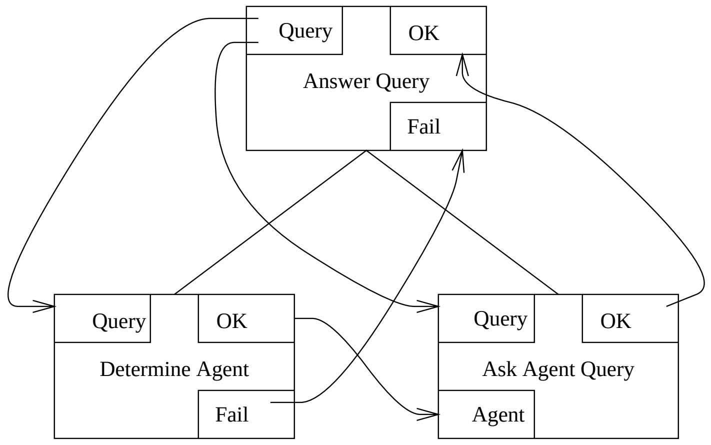

Figure 7: A task reduction for answering a query, showing provision inheritance and outcome disinheritance.

4.3 External provision

We mentioned above that one of our objectives was to support *triggered actions*, which are executed in response to external events. In our system, the most common such event is the arrival of a message from another agent. Our architecture has a mechanism for dynamically routing incoming messages to the provisions of specific actions in the task network. Our agents communicate using KQML [6], and the incoming messages are routed according to the value of their `IN-REPLY-TO` field. Such a routing can be established as part of a reduction schema.

For example, the "Ask Agent Query" task in Figure 7 might be further reduced into two subtasks: one which sends the query to another agent, and one which receives the reply and extracts the answer to the query (Figure 8). The Process-Reply task will be enabled when it is provided with an incoming message.

5 Provisions, parameters, and planning

In a fully-instantiated task network, control flow is derived entirely from information-flow relationships. These relationships are established when the plan is generated. In our hierarchical task reduction planner, each reduction schema defines a set of subtasks and specifies the various provision, inheritance, disinheritance, and external provision links that exist between subtasks and their parent. A reduction may also specify that certain values are to be supplied to certain subtasks at task reduction time. Thus, a task in the network may have its provisions supplied either before or after it is reduced.

Provisions supplied to a task are supplied to the subtasks that inherit them regardless of whether they are supplied before or after the reduction occurs. In Figure 8, for example, the Ask-Agent-Query task could have its QUERY provision supplied after it was reduced, in which case the given value would be immediately forwarded on to the Send-Query subtask. But on the other hand, the value could also be provided *before* Ask-Agent-Query was reduced. In that case, it would be queued there until reduction occured. Upon reduction, any existing

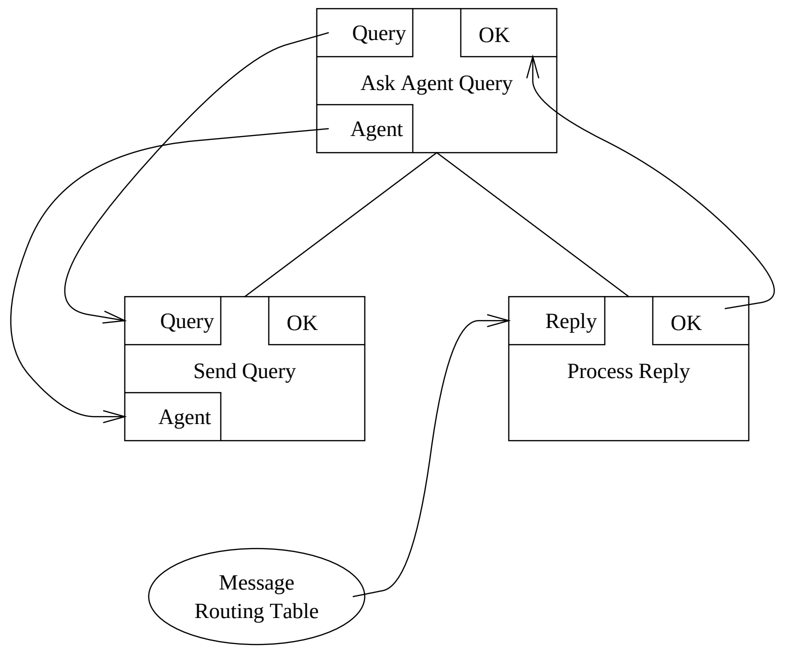

Figure 8: A task reduction for asking a specific agent a query, showing external provision.

provisions are passed on to the inheriting subtasks.

Parameters are handled somewhat differently. A task that has parameters will not be reduced until all the parameters have been provided. This is because the choice of which reduction to use can depend on the values of the task's parameters. This does not mean, however, that all parameters in a task network must be supplied before execution can begin. Our agents generate and execute plans concurrently; any actions can be executed whenever they are enabled, which may occur before a task tree is fully reduced. It is possible for a task to have parameters provided by one of its siblings, in which case planning for that task will be blocked until the task providing the parameter is complete. For example, if AGENT is a parameter to the Ask-Agent-Query task in Figure 7, then that task will not be reduced until the Determine-Agent task has completed execution and provided a result. This would allow the planner to reduce Ask-Agent-Query differently depending on to whom the query is directed.

6 Conclusion

In this paper, we have addressed the issue of control flow and information flow in hierarchical task networks. We have proposed that control flow is derived from information flow relationships. We have presented a framework which unifies and generalizes two existing information-flow mechanisms, runtime variables and outcomes/contexts. This framework supports the representation and execution of task networks with information producing actions, conditional branches, loops, periodic actions, and externally enabled actions. An implementation of this framework is currently in use in the WARREN multi-agent system.

References

[1] J Ambros-Ingerson and S. Steel. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. on A.I.*, pages 735–740, 1988.

[2] K.S. Decker, V.R. Lesser, M.V. Nagendra Prasad, and T. Wagner. MACRON: an architecture for multi-agent cooperative information gathering. In *Proceedings of the CIKM-95 Workshop on Intelligent Information Agents*, Baltimore, MD, 1995.

[3] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, June 1994.

[4] K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, pages 249–254, June 1994.

[5] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, San Francisco, CA, October 1992. Morgan Kaufmann. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[6] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management CIKM'94*. ACM Press, November 1994.

[7] R.J. Firby. Task networks for controlling continuous processes. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, 1994.

[8] R. Goodwin. Using loops in decision-theoretic refinement planners. In *Proc. 3rd Intl. Conf. on A.I. Planning Systems*, 1996.

[9] C. Knoblock. Generating parallel execution plans with a partial-order planner. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, pages 98–103, June 1994.

[10] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. 15th Int. Joint Conf. on A.I.*, pages 1686–1693, 1995.

[11] S. Lin and T. Dean. Generating optimal policies for markov decision processes formulated as plans with conditional branches and loops. In *Proc. 2nd European Planning Workshop*, 1995.

[12] R. Moore. A Formal Theory of Knowledge and Action. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World*. Ablex, Norwood, NJ, 1985.

[13] L. Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, pages 867–874, 1987.

[14] L. Ngo and P. Haddawy. Representing iterative loops for decision-theoretic planning. In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*, 1995.

[15] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. on A.I. Planning Systems*, pages 189–197, June 1992.

[16] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1), February 1994.

[17] D. Smith and M. Williamson. Representation and evaluation of plans with loops. In *Working Notes of the AAAI Spring Symposium on Extended Theories of Action: Formal Theory and Practical Applications*, Stanford, CA, 1995.