HIGH-LEVEL PLANNING IN A MOBILE ROBOT DOMAIN

Technical Note 388

July 15, 1986

By: David E. Wilkins
Artificial Intelligence Center
Computer Science and Technology Division

This paper will appear in a special issue of the *International Journal of Man-Machine Studies.*

The research reported herein is supported by the Air Force Office of Scientific Research under Contract F49620-79-0188.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Abstract

An application of the SIPE planning system to high-level task planning for an autonomous indoor mobile robot is presented. The primary purpose was to evaluate the adequacy of SIPE for this domain, extending and improving the system in the process. The mobile robot domain as encoded in SIPE and the approach to interfacing the planner and the lower-level routines are described. The bulk of the paper presents both problems encountered during the process of encoding this domain, and extensions of the planning system that were made to solve them.

The most significant addition was a redesign of the deductive capability of the planner, which is described in some detail. Efficiency considerations and the ability to intermingle planning and execution are discussed. The most important problem encountered involved *hierarchical planning*, an ambiguous term. We present a definition of it, and examine several of the reasons for this ambiguity. An explication of hierarchical-planning implementations entails two distinct notions: *abstraction level* and *planning level*. A problem in currently implemented planners that is caused by mixing these two levels is presented and various remedies suggested. Three solutions that have been implemented in the current SIPE planning system are described.

1 Introduction

Domain-independent planners developed by AI researchers have not been used in real-world applications. There are several reasons for this, one being that programs tailored specifically for a particular domain will generally outperform a domain-independent system that has the domain encoded in it. However, there are more serious problems. Many assumptions made by these planners curtail the types of domains that can be represented. If the application requires sensory input for monitoring the world, some severe obstacles are encountered in the task of interfacing the planner's representations with the sensory data.

Planning the actions of an autonomous mobile robot is a natural application for planning systems and, moreover, a propitious domain for investigating the above problems. We have therefore applied the SIPE planning system to an indoor mobile robot domain. [1] SIPE [14,15] is a hierarchical, domain-independent planning system that incorporates several extensions of previous such systems. Included among these extensions are a perspicuous formalism for describing operators and objects, the use of constraints for the partial description of objects, mechanisms that permit concurrent exploration of alternative plans, heuristics for reasoning about resources, a deductive capability, and an execution-monitoring module that replans when unexpected events occur.

Our primary purpose was to evaluate the adequacy of SIPE for this domain, extending and improving the system in the process. However, our research also has promise for implementing a working high-level reasoning capability for a robot (though the robot's environment will be fairly restricted). In addition, we have had to address the problem of interfacing the high-level reasoning of the planner and the lower-level control and signal processing needed by the robot. The planner deals with *conceptual* entities by using declarative, propositional representations, while lower-level routines deal with *perceptual* entities and frequently use

[1] The research reported herein is supported by Air Force Office of Scientific Research under Contract F49620-79-C-0188.

more iconic representations. While not solving this difficult and central problem, we have developed a framework within which new developments can be accomodated.

We begin by outlining our approach to the interaction between the planner and the lower-level routines. Following that, we briefly describe the mobile robot domain, as encoded in SIPE, and the plans generated by the system. The bulk of the paper describes both problems encountered during the process of encoding this domain, and extensions of the planning system that were made to solve them. The most significant addition was a redesign of the deductive capability of the planner, which is described in some detail. SIPE's ability to represent a larger class of problems than its predecessors is due, in large part, to its ability to perform deductions that depend upon the state of the world. Thus, the extended deductive capability described here is critical. Another extension described is the ability to intermingle planning and execution.

In addition to extensions of SIPE, a problem that applies to all hierarchical planners was uncovered during this application. This paper contains a general discussion of hierarchical planning that enumerates its many uses in various systems, identifies the aforementioned problem, and presents solutions thereto. We conclude by describing some of the efficiency factors that had to be taken into account in implementing this domain. Such considerations are of prime importance in actually using a planner, but are ignored in most of the planning literature.

2 Integrating Planning with Robot Control Systems

We might view an integrated robot-control system as shown in Figure 1. Current robotic systems contain actuators and sensors, operating in conjunction with controllers that interpret their output, and generate commands according to a program that has been written specifically for the job at hand. Our research on SIPE during the last few years has concentrated on the planning system represented by the upper box in the figure. Little work has been done on the interface represented by the two arrows interconnecting the two larger boxes. There is a wide disparity in the representations and techniques used within each box; representations in the planning box are normally in some form of logic, while representations in the latter are often iconic or procedural [6]. Current perceptual reasoning is usually accomplished by special-purpose systems, such as free-space reasoning programs and vision systems. These problems cannot currently be dealt with by SIPE or, for that matter, by any other high-level planning systems.

Many lower-level control systems have been developed, each of which operates in a fairly specialized domain. These include various vision systems, systems for monitoring ultrasound sensors, programs that compute information from optical flow, programs that plan paths or reason about free space, and the like. They are specialized because they make assumptions about the domain. For example, model-based vision systems assume significant a priori knowledge about the geometry of the objects to be recognized [3]. Systems for analyzing optical flow may assume rigid body motion, or they may assume translation but no rotation in the movement of objects. Significant gains in efficiency can often be obtained by enforcing such restrictions.

The scope of our research does not include the development of lower-level control systems. Our goal is to investigate the utility of SIPE as the highest-level planning system in an integrated robot. Our concern is to develop a system that can work in the short term on simple but useful tasks in a constrained indoor environment, as opposed to an ultimate design for a completely robust, intelligent robot.

In particular, we do not want to commit ourselves to particular representations or designs for the lower-level systems. Instead we are using a heuristic interface language that requires SIPE to plan to a suitably low level before executing actions, and has SIPE accept suitably low-level feedback from other systems. Our intention is for the interface language and its

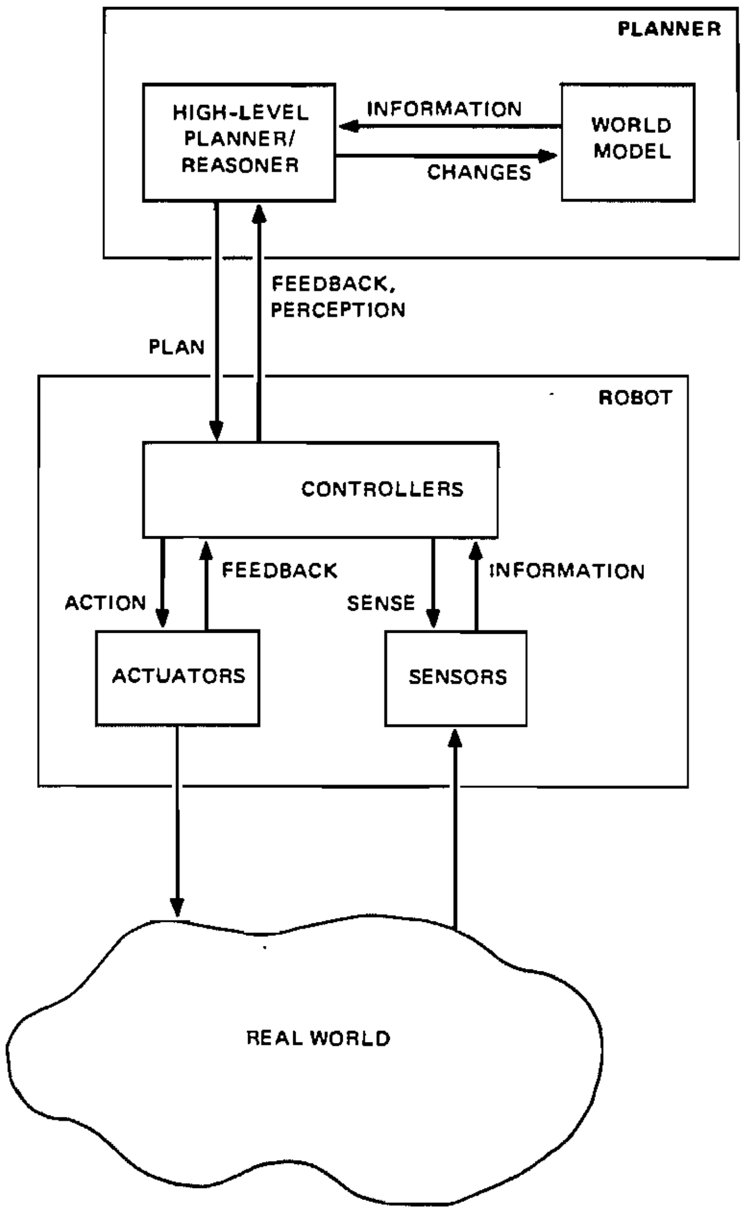

Figure 1: Structure of Robot Control and Planning System

extensions to employ the types of concepts that can be accommodated by diverse low-level systems. The motivation for this is to be able to take advantage of a diversity of research results in these types of systems, rather than committing ourselves to a single approach. For example, instead of presenting a solution to the problem of sensor fusion, we would define reasonable I/O specifications for a system that does fuse information from various sensors, and we would write SIPE operators to interact with these specifications (e.g., speaking of objects being at certain locations with uncertainty bounds). Any system for doing sensor fusion could then be utilized in SIPE after translating between the I/O specifications (a problem which may vary from being easy to being completely unsolvable depending on the particular system).

2.1 Division of Tasks

The central problem is defining the suitable level for the interface including the commands issued by SIPE and the feedback it receives. SIPE is used to generate task-level plans that include information-gathering actions, as well as to replan when unexpected events occur. For example, in planning to get a book for someone, SIPE is responsible for planning how to find the book in someone's office, how to pick it up, and how and where to deliver it. In addition, if the office is inaccessible, SIPE should replan – possibly going to the library for the book, using a different access to the office, or deciding that some other report could replace the book.

Geometric reasoning, precise path planning with obstacle avoidance, and interpretation of sensor data are done by the lower-level controllers, many of which may use iconic representations that are more suitable for these tasks than those of the planner. SIPE plans actions down to the level of actual motor commands for controlling the robot. However, these commands assume that the world is as represented in the planner and that the commands will be executed with complete accuracy. Making these assumptions, SIPE controls a robot simulator in performing tasks, thereby actually giving commands to the robot's motors.

Since neither of these assumptions holds in the real world, low-level routines must use sensors to carry out the actions produced by SIPE. For example, one of the highest-level actions planned by SIPE might be to go forward 50 feet. While this is an actual motor command, SIPE also provides the information that the controller should follow a wall along a hallway to the second door on the right. The controller may cause the robot to veer one way or the other, so as to avoid objects in its proximity as it executes the action of advancing 50 feet. It will also recognize the doorway, thus making the end of the action more precise. In turn, it can report back to SIPE that it has stopped at a particular location (with error bounds) either with success, failure for unknown reasons, failure because an object has been detected at some other particular location, or failure because no doorway has been detected. Given one of these failure reports, SIPE can then replan.

This distributes the planning appropriately, with much of the low-level planning being done outside SIPE. This will also require a special interface for each controller that depends on the representation employed. These interfaces, described in the next section, include the ability to communicate helpful information among modules at any level. While leaving many hard problems unresolved, this technique does permit the SIPE planner to be used while research continues on several approaches to their solutions. The interface with a sonar controller is so far the only one that has been designed in detail. Interfacing with actual programs has been delayed pending their development.

Lower levels may have control for extended periods and, if they send no message back to SIPE during that time, the planner will take no action. The planner will leave the robot under the control of a suitable controller (e.g., one that tries to keep the robot stationary, but may have it dodge objects that are about to hit it) whenever it is actively engaged in planning, thus leaving the robot with at least some ability to cope with the world. SIPE may also pass control to a similar controller if it finds itself confused about the state of the world.

2.2 The Interface

Our design for interfacing SIPE with low-level controllers involves treating them as programmable coroutines. There will be a bidirectional interface language for each controller that will allow the planner to instruct and program the controller, while the latter, in turn, will be able to inform and instruct the planner. The main idea is that the controllers will not be simply subroutines to be called, but will be full-fledged routines, running concurrently with SIPE, that can both send and receive information and requests. Thus, a sensor can monitor the world continually and alert the planner when a certain condition arises. The planner will be able to program such a sensor by telling it what conditions are expected and which unexpected conditions, if they occur, should generate interrupts. For example, when the robot is moving down a hall that is expected to be clear, SIPE wants to be alerted by the ultrasonic sensor if an object is approached, but does not want to receive any messages if the space in front of the robot is clear. On the other hand, if the robot is moving up against a box to push it, the sensor's responses should be reversed.

Our approach allows the planner to instruct the sensors and effectors about what is expected and what should trigger an alert. It also provides for the sensors to request potentially useful information from the planner. One complication is that the instructions and requests that can be supported vary with each low-level controller. In general, a vision system will detect different situations from those that are detected by an ultrasonic sensor or speech-understanding system. Thus, for each controller it will be necessary to develop an interface that accomodates all the useful conditions the controller can recognize for the planner, along with all the types of information the controller can obtain from the planner and utilize in its processing. Furthermore, the types of controllers needed are not readily available, at least commercially.

The planner can help the controllers in many ways. Primarily, it will call these controllers in appropriate situations, given the domain restrictions imposed by the special-purpose algorithms. For example, it could call a specialized path planner only on paths that are expected to be clear. For other paths, the planner would plan to remove obstacles. There are various types of information that SIPE can give these controllers to help them perform efficiently. On the basis of its knowledge of the world, the planner could manipulate some specialized internal representations of a controller. This might be complicated enough to require a special subsystem to generate or manipulate the special-purpose representations of the low-level systems according to predicates supplied by SIPE. For example, the planner can provide a model-based vision system with a list of objects it expects to be in a scene, along with the expected camera angle between the robot and these objects. The vision system can then use this information to greatly reduce the search needed to interpret the scene.

This approach entails a rather loose coupling between the planner and the controllers. This is in contrast to a more integrated system in which all aspects of the systems might be based on the same primitives. Rosenschein's use of situated automata is an interesting example of a more integrated approach [8]. The loose coupling is motivated by the desire to make maximum use of the many specialized algorithms that have been developed. SIPE operators would encode knowledge about when to apply these, what their strengths and weaknesses are, where they can be used most efficiently, and other relevant factors. The planner would then invoke a controller only in situations that match the particular constraints on its algorithm and with suitable a priori information. If SIPE produces reasonable plans, the controllers invoked should be able to carry out their desired functions efficiently.

3 Mobile Robot Domain

The simple indoor mobile-robot world encoded in the SIPE planning system is described here and will be used for expository purposes throughout the rest of this paper. It consists of five rooms connected by a hallway in the Artificial Intelligence Center at SRI International, the robot itself, and various objects. The rooms were divided into 35 symbolic locations that included multiple paths between locations (which greatly increases the amount of work done by the planner). The initial world is described by 222 predicate instances, about half of which were deduced from SIPE's deductive operators. The description of possible actions in SIPE includes 25 action-describing operators and 25 deductive operators. The operators use four levels of abstraction in the planning process. The planner produces primitive plans that provide actual commands for controlling the robot's motors.

The most abstract level of the planning process reasons about the tasks that can be performed, such as preparing a report or delivering an object. Our simple domain requires only one level for this, but the description of more complex tasks might require several abstraction levels. The first level below the task level (referred to as the INROOM level, since INROOM is the crucial predicate) is the planning of navigation from room to room. This plans a route that may require many planning levels for all the necessary operators to be applied, but it does not involve any reasoning about specific doors or locations. High-level predicates describing connections indicate that it is reasonable to move from one room to another, but without first considering any details as to how this might be done or whether it might even be possible in the current situation. When such a move is planned to a lower abstraction level, it may fail or many actions may have to be performed to clear a path.

Below the room level is the NEXT-TO level, which plans movements from one important object (that the robot is next to) to another. For example, to copy a paper, the robot will have to get next to the door of the copy center, then pass through the doorway, then get next to the desk of the operator, etc. This abstraction level plans high-level movements within a room, but is still not concerned with actual locations. NEXTTO is the crucial predicate at this abstraction level.

The lowest level is the location level, where SIPE plans movements down to the level of the actual locational grid it has been given. This may involve planning to move obstacles so as to clear particular paths. AT is the crucial predicate at this abstraction level. The symbolic locations vary in size, but any place of interest could be one. They are small enough so that the controller called to do the path planning can easily solve the problem of getting from one location to the next. For example, each doorway is a location, while long segments of the hall are also single locations. Each location has various attributes within SIPE; among these are the actual two-dimensional extent of the location, the coordinates of a focus, and the coordinates of a focus next to a door, if one exists. From these data the planner can compute the symbolic location of any given real coordinate that might be returned by a sensor.

The predicates used to describe this domain are summarized briefly below. No claim is made that this ontology is optimal in any way – in fact, some predicates were chosen merely to test certain capabilities of the planner. The sort hierarchy includes (among other more obvious classes) BORDERs, which are boundaries between rooms that may or may not coincide with doors, AREAs, which include rooms, halls, and lobbies, and LOCATIONs, which are the symbolic locations described above.

The predicates named below never change value as actions are performed in the world:

(CONNECT BORDER AREA AREA)

These (i.e., all instances of the CONNECT predicate) are given or deduced initially and, for each door, specify which two rooms it connects. They are used to plan room-level paths.

(ADJACENT AREA AREA)

These are all deduced from CONNECT and are used for room-level path planning.

(ADJ LOCATION LOCATION)

These are given or deduced for all adjacent locations (most are deduced). Used for path planning at the lowest level.

(MAILBOX HUMAN OBJECT)

These are all given initially and are used to accomplish DELIVER actions.

(ONPATH LOCATION LOCATION LOCATION)

These give an intermediate location between the two outermost nonadjacent locations and are used for planning a path between the latter.

(DOOR-LOC BORDER AREA LOCATION)

These are all deduced. They tell which location in each room is next to the given door and are used in planning to go through that door.

In contrast to the above group, the predicates named below change as actions are performed:

(AT OBJECT LOCATION)

Most ATs must be given, except that every object that is ON something has AT deduced. AT specifies an object's location at the lowest abstraction level of symbolic locations. Because many deductions are made from it, it is the most important predicate at that level.

(ADJ-LOC OBJECT LOCATION)

These are given or deduced initially for all objects. They tell which locations an object is adjacent to and are used for positioning one object next to another. During planning, these are deduced from ATs. Keeping a current list of ADJ-LOC instances may not be the most efficient way to encode this domain, but it provides a good test of the new deductive capabilities of the planning system.

(INROOM OBJECT AREA)

Most initial INROOMs are deduced. INROOM specifies an object's location at the room level and helps solve problems of abstraction-level coordination (see later discussion on hierarchical planning).

(CONTAINS LOCATION AREA)

Similar to INROOM, but is added dynamically because it is used to coordinate levels of abstraction (see later discussion on hierarchical planning). These are posted as effects involving uninstantiated LOCATION variables, so that the room containing them can be known even though their instantiation is not. INROOM could have been used, but it is clearer to separate the different function performed by CONTAINS.

(OPEN BORDER)

Given initially for borders and specifies whether they are clear.

(ON OBJECT SUPPORT)

These are given initially and trigger many deductions. They are used For noting that an object is on some support (which will determine its location and what it is adjacent to).

(CLEAR LOCATION) (CLEAR BORDER)

All given initially, these specify whether a location is clear enough for the robot to navigate through.

The problem given the planner was to get an object from a corner of a distant room and deliver it to a desk in another room. The planning process required seven levels of planning and took 30 seconds to execute a Symbolics 3600. About half this time was spent processing constraints that kept open (until the very end) the possibility of using either of two alternative paths to the critical locations. There was no backtracking. The planner produced 194 goal/process nodes and 70 additional control nodes in the plan; the most primitive level contained 58 goal/process nodes. 228 plan variables were created during the planning, most of them for deductions. This plan can be executed on our robot simulator, for which it generates actual motor commands intended for the robot. During execution the planner can be told that a path is blocked or a door shut. When this happens, the planner correctly replans to take a different path to the goal location or, alternatively, to procure the object from a different location.

4 Deduction

A number of extensions were added to SIPE to enable the robot-world problem to be encoded efficiently. The most important addition was a complete redesign of the deductive capability to enable more powerful and useful deductions. Domain-independent planners that have used a NOAH-like approach (as distinct from a theorem-proving approach) have not had a deductive capability. In addition to operators describing actions, SIPE allows specification of operators that are used to deduce effects of an action in the current world state. These operators can be viewed as *causal rules* which the system uses to reason about what a particular event causes. An action will generally have many effects that are conditional on the environment in which the action is performed. By representing these conditional effects in causal rules, it is possible to use one action description to cover many situations, thus reducing the number of actions the system must consider in the planning process.

As more complex domains are represented, it becomes increasingly important to deduce the effects of actions from causal rules about the world, rather than representing these effects explicitly in operators. Besides being necessary for execution monitoring, deduction is important for determining both side effects and conditional effects of an action. This capability, which distinguishes SIPE from its predecessors, is the primary reason the extension described herein significantly expands the ability of the system to deduce formulas.

Since deductive operators in SIPE have already been described in some detail [14], we shall summarize them briefly. The following subsections give detailed descriptions of the recent extensions. SIPE maintains strict control over the application of deduction so as to prevent a combinatorial explosion, while still providing a powerful enough capability to be useful. All deductions that can be made are performed at the time an operator is expanded. The deduced effects are recorded in the procedural net and the system can then proceed just as if all the effects had been listed in the operator. Deductions are not attempted at other points in the planning process. Deductive operators have triggers to control their application.

If the precondition of a deductive operator holds, its effects can be added to the world model (in the same context in which the precondition matched) without modifying the existing plan.

The two major changes in the system's deductive capability are the ability (1) to perform chains of deductions and (2) to encode deductive operators by using "universal" variables. The former enables many more deductions to be made by the repeated application of deductive operators. The latter permits the expression of more powerful deductive operators.

4.1 Deductive Chains

Initially, only deductive operators whose trigger predicate matches a node-specified effect are applied, thereby producing an additional set of [deduced] effects for that node. SIPE previously stopped the deductive process at this point; only one-step deductions were allowed, with the process kept under strict control and deductive loops precluded. SIPE has been extended to allow deductive chains of arbitrary length. After all deductive operators have been applied, the system determines which newly deduced effects were not already true in the given situation and permits the deductive operators to trigger on these recursively. This process continues until no effects are deduced that were not already true. This deductive capability allows more powerful deductions while maintaining control of the deductive process and preventing deductive loops.

There is one problem in the foregoing scheme. Namely, the possibility exists that a later deductive operator might deduce a predicate that negates a predicate deduced by a previous operator. Which of these conflicting deductions should the system allow to stand as an effect? In general, deductive operators should be written to avoid this kind of conflict. However, there is a situation, which was first observed in the mobile robot domain, in which it is desirable to deduce effects that would be conflicting if they were to be matched directly against each other, but become a valid non-conflicting representation of effects when they are recorded in the order they are deduced. This situation, described in detail below, involves the deduction of a predicate with a "universal" variable (defined in the next section) that negates particular nonuniversal instances of the predicate that have already been deduced.

This can be useful because the matching algorithm in SIPE matches formulas against effects in the order they are listed. Thus, one can initially deduce (ON OBJECT1 OBJECT2), where the objects are not universals (they may be variables that are not yet instantiated but eventually will be to one particular object), and later deduce (NOT (ON OBJECT1 OBJECT3)), where OBJECT3 is universal. These will be recorded in the given order which effectively encodes the fact that OBJECT1 is on OBJECT2 only. More precisely, any formula of the form (ON OBJECT1 X) will match positively if X can be constrained to be the same as OBJECT2, negatively in all other cases. If conflicting deductions are made in other cases, SIPE provides user-selectable options of signaling an error, recording the first deduction, or recording the second deduction, with all choices terminating the deductive chain. The user can choose whichever action is most appropriate in his domain.

4.2 Universal Variables

SIPE fomerly permitted universal variables only when they occurred in the effects of an operator, but has now been extended to allow the presence of "universals" in preconditions of deductive operators. The use of universals in preconditions involves a major change in the basic matching process of the planning system.

The semantics of a universal in a SIPE precondition differs significantly from the use of universal quantifiers in formal logic (hence the previous quoting of "universal"). Let us suppose that X is a universal variable in P, a precondition. This does not mean that the system must prove the precondition true for all possible instantiations of X (i.e., $\forall x.P(x)$), rather it means that only instantiations such that P is true will be considered for X heretofore (i.e., $x \mid P(x)$). During matching of such a precondition, the system will generate constraints enabling the variable to match all and only those objects for which the precondition is true.

This capability to form subsets is very useful and powerful. It exploits SIPE's representation and algorithms for the purpose of representing a number of predicates compactly as a single predicate with a universal variable. A primary advantage of this is the gain in efficiency that is achieved by matching the deductive operator only once, instead of having to match it once for each possible match of the universal variable.

For example, suppose we wish to deduce something about all the blocks that are on a table. The precondition of a deductive operator can specify (ON BLOCK1 TABLE), where BLOCK1 is a universal variable. When SIPE matches this formula, it will post constraints on BLOCK1 that allow it to match only those blocks that are on the table. This effectively picks out the subset of blocks that interests us and allows their efficient representation. The constraints specify that BLOCK1 must match one of a set of N planning variables. Some of these variables may not yet be instantiated, but eventually they will. Perhaps they will be instantiated to N different blocks, or they may all be instantiated to the same block. There is no problem matching because the constraints on each of the N planning variables specify all relevant information (e.g., which variables and blocks these N planning variables are/aren't identical to).

The further planning of actions occurring either earlier or later will not affect the validity of the universal variable in the ON predicate. This is true by virtue of the place where the predicate is recorded in the plan as well as because of the matching algorithm employed by SIPE [14]. Suppose N1 is one of the N planning variables that will match the universal variable. If a later action specifies (NOT (ON N1 TABLE)) as an effect, this latter predicate will always be matched to a formula before the predicate with the universal variable (since SIPE regresses through the plan attempting to find matches). Thus, the appropriate relationship between N1 and the corresponding variable in the formula being matched will already be determined and will not be affected by any subsequent attempt to match the same formula with the predicate containing the universal variable. Further planning of an action before the one containing the ON predicate as an effect will be done at the next lower [hierarchical] planning level (as defined in next section). In this case, SIPE recomputes all deductions that follow this one at the next planning level. Thus, a new universal variable will appear with constraints that have been properly calculated for the new situation.

This use of universals involves a major change in the matching algorithm used by SIPE to determine the truth of a predicate. Before this extension, a universal variable matched any other variable in its class exactly. Consequently, the matcher had to look no further for possible matches of a predicate. Allowing constraints on universals means that now they no longer match exactly, but are instead only potential matches that depend on the instantiation of variables in the predicate being matched. The matching process must therefore continue collecting other potential matches and let the system generate additional constraints upon the possible ways the predicate could be made true.

5 Hierarchical Planning

Another problem that arose during encoding of the robot domain involved hierarchical planning levels. The task of describing this problem involves explicating the considerable ambiguity involved in hierarchical planning, as well as distinguishing two distinct notions: *abstraction level* and *planning level.*

It is generally recognized that planning in realistic domains requires planning at different levels of abstraction [5]. This allows the planner to manipulate a simpler but computationally tractable theory of its world. The combinatorics of concatenating the most detailed possible descriptions of actions would be overwhelming without the use of more abstract concepts. This has resulted in numerous hierarchical-planning systems. However, hierarchical levels and hierarchical planning mean quite different things in different planning systems, as is explained in the next section.

In our view, the essence of hierarchical planning (and a necessary defining condition) is the use of different levels of abstraction – both in the planning process and in the description of the domain. An *abstraction level* is distinguished by the granularity [5], or fineness of detail, of the discriminations it makes in the world. From a somewhat more formal standpoint, a more abstract description (in whatever formalism is being used) will have a larger set of possible world states that satisfy it. When less abstract descriptions are added, the size of this satisfying set diminishes as things in the world are discriminated with increasingly finer detail. In complex worlds, these abstract descriptions can often be idealizations. This means that a plan realizable at an abstract level may not be realizable in a finer grain (i.e., the satisfying set might reduce to the null set). For example, one might ignore friction in an abstraction of the domain, yet find that the abstract plan cannot be achieved at the lower abstraction level when the effects of friction are included in the world description.

To see how hierarchical planning can help avoid the combinatorial explosion involved in reasoning about primitive actions, let us consider the task of planning to build a house. At the highest abstraction level might be such steps as site preparation and foundation laying. The planner can plan the sequence of these steps without considering the detailed actions of hammering a nail or opening a bag of cement. Each of these steps can be expanded into more detailed actions, the most primitive of which might be nail-driving and wire-cutting. Hierarchical abstraction levels provide the structure necessary for generating complex plans at the primitive level.

5.1 The Many Guises of Hierarchical Planning

The planning literature has used the term "hierarchical planning" not only to describe levels of abstraction, but also to describe systems containing various hierarchical structures or search spaces, metalevels, and what we shall call *planning levels.* Examples of each of these are given below. Planning levels are of particular importance because confusing them with abstraction levels causes a problem in various implementations of hierarchical planning that will be discussed in the remainder of this paper.

Many planners produce hierarchical structures (e.g., subgoal structures) during the planning process or explore hierarchically structured search spaces. Generally having nothing to do with abstraction levels, they occur even in nonhierarchical (by our definition) planners that allow only one level of abstraction. STRIPS [1], while nonhierarchical, could be regarded as producing plans with a hierarchical structure, e.g., its triangle tables. The Hayes-Roths [4] use the term *hierarchical* to refer to a top-down search of the space of possible plans in which more abstract plans are at the top of this space. This involves a hierarchical search space that contains abstraction levels; the latter, however, do not define the levels in the hierarchy.

Hierarchical planning is also used to refer to metaplanning. Reasoning at a metalevel involves reasoning about the planning process itself. This is an entirely different domain, not merely an abstraction or idealization of the original domain. Stefik [11] states that ". . .layers of control (termed *planning spaces*) . . . are used to model hierarchical planning in MOLGEN". In this case, the three planning spaces are being used to implement metaplanning and, respectively, represent knowledge about strategy, plans and genetics; the first two are not abstractions of the genetics domain. (MOLGEN does provide for planning at different levels of abstraction through its constraints.)

The above instances of the term "hierarchical planning" describe processes or structures that are unrelated to the use of abstraction levels. Therefore, any confusion generated is terminological and does not reflect possible conceptual problems within the planning system itself. Planning levels, on the other hand, have been confused with abstraction levels – which, as we shall see, can lead to problems within the planner, particularly if the planner incorporates the STRIPS assumption [13]. *Planning levels* are artifacts of particular systems and may vary considerably from planner to planner. They are not defined by a different level of abstraction in the descriptions being manipulated, but rather by some process in the planning system. Most systems have a central iterative loop that performs some computation on the plan during each iteration. This may involve applying schemas, axioms, or operators to each element of the existing plan to produce a more detailed plan. To the extent that such an iteration takes one well-defined plan and produces another well-defined plan, we will call it a planning level. In some systems, planning levels may correspond exactly to the hierarchical structures discussed earlier, but this is purely coincidental. They are defined by the planning process, not data structures, and may or may not correspond to hierarchical data structures within a particular system. The term is admittedly vague, but in many AI planning systems of interest it nevertheless has a very precise definition.

In particular, all planning systems in the NOAH tradition, including SIPE, NOAH [10], NONLIN [12], and ABSTRIPS [9], have distinct and well-defined planning levels. In these systems, a new planning level is created by expanding each node in the plan with one of the operators that describe actions. In the literature these levels are often referred to as hierarchical, which implicitly associates them with abstraction levels. The fact is that they are independent of abstraction level; a new planning level may or may not result in a new abstraction level, depending upon which operators are applied. For example, in the blocks world described by Sacerdoti [10], there is only one abstraction level. CLEARTOP and ON are the only predicates and each new planning level simply specifies in greater detail a plan involving these predicates. Thus, all the hierarchical levels in the NOAH blocks world are actually planning levels that result in adding further detail to the plan at the same abstraction level. Others have described such a prior omission of detail as an "abstraction", but we specifically require an abstraction level to involve different predicates with different grain-sizes.

Planning systems not in the NOAH tradition also have planning levels. Rosenschein's planning algorithm, using dynamic logic [7], attempts to satisfy a set of planning constraints. The result of running his "bigression algorithm" on each constraint in the set (which, he contends, is a straightforward extension of his system) would constitute a planning level. Agenda-based planning systems also have natural planning levels that are defined by the execution of one agenda item. These planning levels would be somewhat different from the others we have discussed, as they might not involve performing some operation on each element of a complete plan. Consequently, they are not likely to be confused with abstraction levels.

5.2 A Problem Encountered with Current Planners

There is a problem that can arise when planning and abstraction levels are interleaved in a planner making the STRIPS assumption, as they are in the aforementioned systems. (The STRIPS assumption states that things remain unchanged during an action unless specified otherwise.) This problem exists in many NOAH-tradition planners but has never been documented.

In such a planner, one element of the plan can attain a lower abstraction level than another element at the same planning level, depending upon which operators are applied. Thus, the plan at the current planning level could be **P1;Q1;G**, where **P1** is an action making the predicate P true, **Q1** an action making the more abstract predicate Q true, and **G** a goal that depends upon the truth of P for its achievement. With the STRIPS assumption, the planning system will find that P is true at **G**, since **Q1** does not mention changes involving any less abstract predicates (such as P). In fact, the truth of P may depend upon how **Q1** is expanded to the lower abstraction level, since it may or may not negate the truth of P. Thus, conditions may be evaluated improperly in these planners, resulting in incorrect operator applications.

For these planners to test the truth of a condition correctly at time N in the plan, they must ensure either that all relevant information at the proper abstraction level is available for the actions preceding N or that subsequent expansions to lower abstraction levels will not change the truth-value of the condition. However, many existing planners (e.g., NOAH and SIPE) do not provide this assurance. There are good reasons for this. Various solutions to the above problem are discussed later in this paper, but the most straightforward one is to impose a depth-first, left-to-right planning order that is sensitive to any change in abstraction level. This means that, when a condition with predicates of abstraction level M is checked at time N in the plan, all possibly relevant information at abstraction level M or higher will be available for every plan element occurring prior to N.

However, this is not always desirable, since many advantages can be gained by planning certain parts of the plan expedientially (or opportunistically) to lower abstraction levels. For example, in planning a trip from Palo Alto to New York City, it might be best to plan the details of the stay in New York first, as this could determine which airport would be best to fly into, which in turn could determine which Bay Area airport would be the best departure point. Thus, we do not want to restrict ourselves to depth-first, left-to-right planning, which would require choosing the Bay Area airport before the one in New York.

While the foregoing problem will be discussed in this paper in the NOAH-like terminology of "operators" and "goals", it applies equally well to any planner that must coordinate deductions over different abstraction levels. For example, Rosenschein's hierarchical planner based on dynamic logic [7] must address this issue in order to produce correct plans.

The definition of the problem will be made more concrete by viewing it in the context of an indoor robot domain. The problem of coordinating abstraction and planning levels arose during the planning depicted in Figure 2. The initial goal produced three INROOM goals at the first planning level. Of these three subtrees, the first and last were transformed into the lowest abstraction level at planning level 2 (intervening NEXTTO goals have been disregarded for the sake of simplicity). The middle subtree, however, is still at the room abstraction level on planning level 3, as this subtree required several operator applications to find a path through the rooms to accomplish its goal.

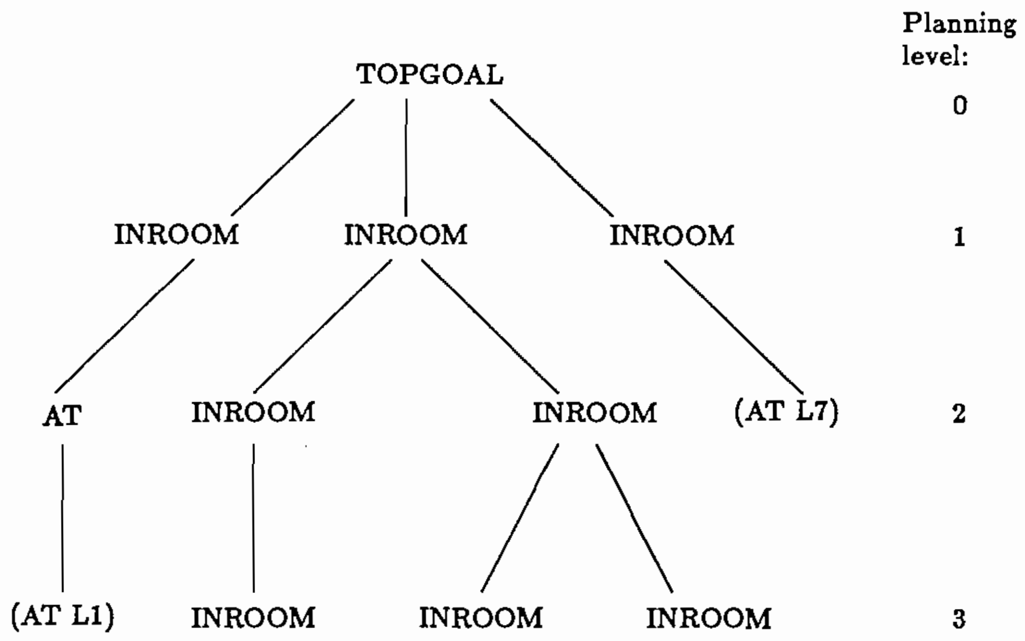

Figure 2: Hierarchical Plan in Robot Domain

Now, when the planner applies an operator to the (AT L7) goal at planning level 2, any precondition that queries the AT predicate will find (AT L1) to be valid, since that is the last place at which AT is affected and the STRIPS assumption assumes that all actions leave predicates unchanged unless they explicitly indicate otherwise. But this is not correct, as AT may be affected after the middle subtree has been expanded to the lowest abstraction level. If the operator depends critically on the value of the AT predicate, its application to (AT L7) may be incorrect. The resulting plan will have commands that move the robot from location L1 to location L7, whereas the robot is not likely to be at L1 when that part of the plan is executed (because of movements made in the middle subtree). Whether such an operator application will prevent the correct plan from being found depends on how a particular planner detects invalid plans and how the search space is organized.

5.3 Solutions

There are many alternative solutions to the problem discussed above. They range from calculating all possibly relevant information before it may be needed to ignoring the problem altogether and simply letting the user beware of the consequences. The former – and most straightforward – is to force the planner to plan in temporal order and to provide for calculation of all the predicates at one abstraction level that may be needed later. Many planners do this, though it is usually not made clear that they depend on several assumptions to avoid subtle problems such as this one. ABSTRIPS [9] is an example of planners that employ this approach. It assigns abstraction-level numbers to the predicates and plans a lower level only when all necessary computations have been made. SIPE provides, as a user-selectable option, the ability to control hierarchical planning in this manner. This is useful for comparing the performance of the techniques described below.

The problem with this approach is that the planner is limited as to the sequence in which it can process goals. Quite often the order imposed will not be optimal (as in the Palo Alto-to-New York example). The flexibility to plan certain parts of the plan expedientially to lower abstraction levels is lost. Constraints generated during such lower-level planning can narrow down the search, thus resulting in potentially large gains in efficiency. For these reasons, planners like SIPE, NOAH, and NONLIN allow the mingling of planning and abstraction levels.

The approach used in these latter planners is susceptible to the level coordination problem and therefore requires the user to be alert. Erroneous checking of conditions similar to the one described can occur in these systems, depending on the task and the encoding of the operators. This can result in incorrect operator applications. The planning system may have mechanisms that subsequently detect incorrectness of the plan, as SIPE and perhaps NONLIN do. However, the proper solution may not be found if the operator applied incorrectly is the one actually needed for the correct solution, since it may not be retried. Its premature application should have been delayed one or more planning levels so that other parts of the plan could be planned to a lower abstraction level. The user is responsible for writing operators that will accomplish this delay. This process is facilitated by certain features of SIPE, as we shall see.

It is appealing to look for a technique between the inefficiency of computing everything in a certain order and the expediential behavior of systems that perhaps miss valid solutions. This would involve reasoning about what properties will remain invariant during the further planning of certain goals. For example, in Figure 2 the planner might calculate whether the possible expansions of the middle subtree will affect the value of the AT predicate. If not, planning can proceed expedientially. This is all the more appealing because reasoning about concurrency depends on determining invariant properties of a sequence of actions.

Despite the attraction of this approach, there are severe difficulties entailed in computing these invariances. It would be best if they could be computed automatically from the operators without requiring the user to supply additional knowledge. (The STRIPS assumption effectively makes this computation for predicates at one abstraction level, but the computation must be done for predicates at other abstraction levels.) In general, however, this is not computationally tractable. It is similar to the problem of regressing conditions through actions ([7],[13]), except that the regression must be done through every possible expansion of the actions for an indeterminate number of planning levels. Furthermore, simple schemes that merely check for possible changes in predicate names will probably find very few invariances. In the robot domain, for example, every high-level goal will alter the values of AT predicates at the lowest level. More sophisticated schemes that check possible values for the arguments of the predicates, perhaps determining the ranges of values they might acquire in all possible expansions, would themselves necessitate solving a large search problem.

An alternative is to have the user provide information about what remains invariant over actions. While this may be useful for some domains, in general the same criticisms made

```
OPERATOR: not-yet
ARGUMENTS: robot1,location2,area1,location1;
PURPOSE: (at robot1 location2);
PRECONDITION: (at robot1 location1),
      (inroom robot1 area1),
      (not (contains location1 area1));
PLOT: COPY
END PLOT END OPERATOR
```

Figure 3: Operator for Delaying Operator Application

above apply to this case. There will generally not be many things (at lower abstraction levels) that are invariant; moreover, it may even be difficult to indicate explicitly what they are, as the invariance may involve complex constraints on the allowable arguments to predicates. In addition, one of the chief advantages of abstraction levels is that specifying details is unnecessary at higher levels. Computing invariants would require information as to which lower levels are affected in what way by each higher-level goal, thus removing some of the advantage gained by not planning at the lowest level from the very beginning. The computational costs of this can quickly become overwhelming, as we shall see in the next section.

5.3.1 Delaying Operator Applications in SIPE

For reasons given above, SIPE allows the user to assume the burden of encoding the domain in such a way that incorrect evaluation of conditions will not produce applications of operators that prevent solutions from being found. We have solved this problem in two ways within SIPE for the indoor robot domain (in addition to the option of using the ABSTRIPS solution). One solution involves delaying the application of certain operators while the other entails the introduction of certain less abstract predicates at earlier planning levels.

The first solution involves a novel use of operators, developed during implementation of the robot domain, that effectively defers the achievement of certain goals until the appropriate juncture, as long as the latter can be ascertained by conditions that are expressible as preconditions of a SIPE operator. We can best show this by reintroducing the robot domain example.

In the robot domain, only the planning of AT goals is affected when abstraction levels vary in the plan. Figure 2 depicts the type of situation in which the accomplishment of an AT goal must be delayed. This is done in SIPE by using the operator shown in Figure 3. It postpones the solving of AT goals until the part of the plan preceding them has been brought to the same level of abstraction. This is done by checking whether the AT location of the robot is in the same room as its INROOM location. If the precondition of this operator matches, it means that the last AT predicate specified as an effect of an action came before the last INROOM predicate specified as an effect. Consequently, the latter action must still be planned to the lower level of abstraction. [2]

This operator is applied before any other to an AT goal. The plot (i.e., consequent) of not-yet is simply the token *COPY* that copies the goal from the preceding planning level. It is necessary to use a special token rather than specify the AT goal in normal syntax. Normally SIPE inserts the precondition of an operator into the plan and maintains its truth. In this case the precondition will not be true in the final plan, so the COPY option inserts the appropriate goal without first inserting the precondition. With this feature and the above operator, SIPE can mix abstraction and planning levels freely in the robot domain without missing a solution to our test problems. Nonetheless, there may still be problems in the domain that cannot be solved without creating additional delaying operators.

[2] Of course, this operator could still be fooled if one planned a circular route that ended in an INROOM goal for the same room that contained the preceding AT location. However, this would cause a problem only if the eventual location reached in the expansion of the INROOM goal were different from the one in the earlier AT goal. This situation never arises in our domain.

5.3.2 Introducing Low-Level Predicates in SIPE

Instead of using the not-yet operator, the second solution involves introducing lower-level predicates at higher abstraction levels to prevent the STRIPS assumption from causing a problem. In this case, we add a lower-level AT predicate (which includes an uninstantiated locational variable) to every higher-level NEXT-TO goal as a placeholder for the predicate that would be produced during some future expansion. When the NEXT-TO goal is expanded to the lower level, the location actually reached will eventually become the instantiation of the locational variable introduced. At the planning level of the NEXT-TO goal, any AT predicate in a condition being tested will match the newly inserted AT predicate, thus preventing the planner's incorrect assumption that the NEXT-TO goal does not affect the truth-value of the condition.

This solution takes advantage of SIPE's ability to post constraints on variables. The newly introduced AT predicates effectively document the fact that the AT location may eventually change during any expansion of the NEXT-TO goal (even though the location is not yet known). Before and during such an expansion, the locational variables can accumulate constraints on their possible values, so the planning process will not be hindered. The matching of conditions will always be correct because these AT predicates are present wherever the AT location might change.

Incorporating this change into the SIPE operators written for the first solution was easy. Only three operators of the 25 posted NEXT-TO goals, so only those three had to be changed. The process of converting these three operators is illustrated by contrasting the original FETCH operator shown in Figure 4 with the FETCH operator including AT predicates that is shown in Figure 5. In the plot, each goal and process with a NEXT-TO predicate in its effects is given an additional effect that is an AT predicate involving a new locational variable. The latter is included in the arguments of the goal or process so as to permit appropriate matching with the variables in the operators that solve NEXT-TO goals. The locational

```
OPERATOR: fetch
ARGUMENTS: robot1,object1,area1;
PURPOSE: (holding robot1 object1);
PRECONDITION: (inroom object1 area1);
PLOT:
    GOAL: (inroom robot1 area1);
        MAINSTEP: (holding robot1 object1);
    GOAL: (nextto robot1 object1);
    PROCESS
        ACTION: pickup;
        ARGUMENTS: robot1, object1;
        EFFECTS: (holding robot1 object1);
END PLOT END OPERATOR
```

Figure 4: Original FETCH Operator

```
OPERATOR: fetch
ARGUMENTS: robot1,object1,area1,location1;
PURPOSE: (holding robot1 object1);
PRECONDITION: (inroom object1 area1),
    (contains location1 area1);
PLOT:
    GOAL: (inroom robot1 area1);
        MAINSTEP: (holding robot1 object1);
    GOAL: (nextto robot1 object1);
        ARGUMENTS: robot1, object1, area1, location1;
        EFFECTS: (at robot1 location1);
    PROCESS
        ACTION: pickup;
        ARGUMENTS: robot1, object1;
        EFFECTS: (holding robot1 object1);
END PLOT END OPERATOR
```

Figure 5: FETCH Operator with AT Predicates

variable is added to the arguments of the operator, whose precondition can also specify any predicate that constrains the variable. As a minimum, the variable can be constrained by the room in which it is located; stronger constraints can sometimes be specified.

Once the three operators had been converted in this manner, SIPE was able to solve all the problems in our test domain. This solution appears robust and should not prevent the planner from dealing successfully with any problems it might solve with an ABSTRIPS-like approach. Characteristics of the domain are again exploited in this solution. By having to plan about less abstract entities at a more abstract level, we are giving up some of the advantage gained by planning hierarchically. However, it is reasonable to do so in this case because we need introduce only one lower-level predicate early (albeit the most important one), and we need introduce it only a single abstraction level early. There is no difficulty in coordinating any other pair of abstraction levels in this domain. However, the introduction of more variables and constraints increases the effort required significantly. Using this technique to solve problems in the robot domain takes from an additional one-third to twice as long as using the not-yet operator (see next section).

5.4 Comparison of Solutions

The two techniques described above were tested on three problems in the robot domain. The ABSTRIPS control regime was also used to solve these problems. (The operators were the same as those utilized while delaying operator application, except that the not-yet operator was eliminated.) The original robot problem involved a choice of different paths and entailed seven planning levels for producing a primitive plan with 58 process/phantom nodes. The other two problems are similar – but one requires a shorter path to be found, the other a longer path. Figure 6 depicts the CPU time and number of planning levels required for each problem. It is no surprise that the ABSTRIPS control regime is as efficient as delaying operator application, since the solution to these particular problems cannot be simplified by

Problem:	NOT-YET	introduce AT	ABSTRIPS
Original	29.5 (7)	54.2 (7)	32.1 (11)
Shorter path	24.8 (7)	46.7 (7)	25.1 (8)
Longer path	43.7 (7)	57.4 (7)	35.7 (10)

Figure 6: Symbolics 3600 CPU Time and Planning Levels for Solutions

planning later parts of the plan to a lower abstraction level. For problems and domains with this property, ABSTRIPS-like coordination of levels is preferred because it is both efficient and correct. (It does, of course, require more planning levels.)

The first solution (not-yet) accomplishes the delayed application of operators when necessary, but permits expediential planning in other cases. This solution retains flexibility while remaining efficient by not regressing conditions through possible expansions of actions. As no lower-level predicates are introduced early, full advantage is taken of hierarchical planning. The disadvantages of this approach are that the user (though relieved of the necessity of specifying invariance properties for higher-level goals) must write appropriate delaying operators and, furthermore, must anticipate all possible situations in which operators would need to be delayed. In complex worlds, this means that novel problems might not be solvable. In addition, it may not always be possible to express the appropriate delaying conditions as a SIPE precondition. In an application in which efficiency is of paramount importance and failure to solve a particular problem can be tolerated, this may be a desirable approach.

The second solution (AT) is more robust and less likely to fail on novel problems. It was surprisingly easy to implement in SIPE. However, when low-level predicates are introduced at a higher abstraction level, it is significantly less efficient. The advantages of hierarchical planning can be readily observed, as the introduction of only one predicate (albeit the crucial one) at the next higher abstraction level nearly doubles the cost of computation.

6 Intermingling Planning and Execution

Planners in the NOAH tradition have historically planned every step of each plan to the lowest level of detail. This is the reason the plan described in the last section took over 30 seconds to generate. Such detailed planning can often be undesirable [2]. It can take a long time, preventing the planner from reacting quickly to events. Furthermore, as actions are planned farther into the future, it becomes less likely that they will be useful. The probability increases that some unexpected event will render the remainder of the plan unsatisfactory. (In our interface, the robot is under the control of a low-level program during the planning process, so it has some capability to react to events.)

Fortunately, there is no inherent reason that NOAH tradition planners have to plan everything to the lowest level of detail, so SIPE has been extended to intermingle planning and execution. This was a fairly simple task. SIPE's operator description language now allows users to encode domain-dependent information about which goals and actions can have their planning delayed. The user can simply include the word "DELAY" in the description of a node in the plot of an operator. The system will then not plan any such goal or action until a plan suitable for execution has been generated. The planning of the delayed goals is started as a background job as soon as the original plan is ready for execution.

The original plan is used by the execution monitor until either an unexpected event happens or the goals whose planning has been delayed are reached. In both cases, SIPE retrieves the plan produced by the delayed planning process (possibly waiting for the process to finish), and updates it with information about nodes that have already been executed. Execution proceeds on this updated plan while another background job continues to plan any delayed goals in this new plan. When SIPE attempts to retrieve the results of the delayed planning process, it may notice that the delayed planning fails, in which case the system tries again to solve the original problem.

The encoding of domain-dependent knowledge for this purpose is effective because such knowledge is generally available. For example, in the robot problem previously described, the robot can obviously begin executing its plan to get to the object to be picked up before planning what to do after picking up the object (assuming the robot does not make hallways impassable as it travels down them). Thus, the operator for fetching and delivering an object had a delay put on its second goal. Goals should not be marked for delayed planning unless there is a high probability that they can be achieved, or it is known that their solution is independent of the solutions chosen for prior goals. The planner can begin execution with some assurance that its initial plan should be the beginning of a valid solution for the whole problem. Domain-independent criteria for delaying planning, e.g., delaying planning after a certain number of actions have been planned, would be arbitrary and would not be able to provide this assurance.

The delay described above (on the deliver goal following the fetch) is the only one introduced into the SIPE operators used to solve the problems described in the previous section. With this minor addition, SIPE produces a plan for the same problem that is ready for execution in only 9 seconds (as opposed to 35). The remainder of the plan is usually filled out in complete detail before the robot gets very far down its first hallway. This enables SIPE to react much more quickly to situations, and reduces the time spent waiting on the planner. It is also easy to envision more options than simply planning delayed goals in a background job. On the basis of domain-dependent knowledge, they could alternatively be planned immediately or left unplanned until execution reaches that point in the plan.

7 Efficiency Considerations

Efficiency considerations affect both the manner in which a domain is encoded in SIPE and the design of the system itself. The following subsection describes issues that must be addressed during encoding (in particular, of the robot domain). In addition to the complete redesign of the deductive capability already described, a number of extensions were added to SIPE to enable the robot-world problem to be encoded efficiently. Most of these are of such a technical nature that they will not be mentioned here, but a few examples will be described in order to illustrate the kind of detail that must be dealt with to produce an effective implementation. Much of the theoretical work done in planning has simply ignored these practical issues.

7.1 Constraint Efficiency

Constraint efficiency is a problem that arose during the encoding of the robot domain. It is fairly easy to write operators that will produce constraints that are computationally too expensive. The person writing the operators must therefore be careful to formulate things in such a way as to ensure that this computational cost will be reasonable. We encountered this problem during deduction of the ADJ-LOC relationships. This predicate denotes all the locations that are adjacent to a given object. In SIPE, the deduction of this relationship after an object has moved involves both eliminating the old ADJ-LOCs that no longer hold (which is accomplished by adding new negated ADJ-LOCs that will match future conditions before the old ones) and adding the new ones that have now become true.

The problem surfaced during removal of the old ADJ-LOCs that were now invalid. It seemed natural to write a deductive operator that deduced a negated ADJ-LOC for every location that was not adjacent to the object's new location. This works fine when the location being moved to is instantiated; when it is not, however, the number of things that can be not adjacent to it is enormous. A constraint is added to the variable in the negated ADJ-LOC effect in SIPE that must later be processed frequently during the matching required to determine the truth-value of predicates. The computation involved in processing this constraint slowed the system down unacceptably.

Fortunately, we were able to find an elegant solution in SIPE by making use of the

```
DEDUCTIVE.OPERATOR: updateadj
ARGUMENTS: object1,location1,
     location2 is not location1,
     location3 is not location1 class universal,
     location4 is not location3 class universal;
TRIGGER: (at object1 location1);
PRECONDITION: (at object1 location2),
     (adj location1 location3);
EFFECTS: (adj-loc object1 location3),
     (not (adj-loc object1 location4));
END DEDUCTIVE.OPERATOR
```

Figure 7: Operator for Coordinating Levels

universal variables and the NOT-SAME constraint supplied by the system. The removal of invalid ADJ-LOCs is now done by the same operator that deduces new ADJ-LOCs; the operator simply uses a universal variable that is NOT-SAME with the universal variable in the new ADJ-LOCs. In other words, if a location is not the same as an ADJ-LOC of the new location, then it is not an ADJ-LOC of the new location. The operator that deduces this is shown in Figure 7 as it is inputted to SIPE.

7.2 Practical Considerations

To illustrate the practical issues that must be resolved in an implementation, this section describes two of the many extensions added to SIPE to improve efficiency. The first involves the use of NOT-SAME universal variables, as shown in Figure 7. It has proved useful in the mobile robot domain to deduce all positive as well as negative occurrences of a certain predicate using universal variables for both deductions. Once the proper constraints have been generated for the universal variable in one of these deductions (whether it be the positive or the negative one), the universal variable in the other deduction can simply be constrained to be not the same as its counterpart. Since this appears to be a generally useful technique, SIPE includes a check for such NOT-SAME universals so as to make the matcher more efficient.

When looking for potential matches of a predicate, SIPE checks to determine whether two possible matches are negations of each other, with exactly the same arguments save one. If this one argument is a universal variable in both possibile matches, and if the variables are constrained to be NOT-SAME with each other, then the system knows that all possibilities have been covered and so ends its search for matches. The fact that this application of NOT-SAME universals is of general utility justifies inclusion of the foregoing check. This check for NOT-SAME universals exemplifies the details that must be considered if an implementation is to be efficient.

A second efficiency consideration involves the amount of checking and posting of constraints that is done during the formula-matching process. As the matcher accumulates possible matches, it may generate constraints based on them that it would then propagate while checking their interaction with constraints on other variables. This would be expensive, as maximum effort would be invested to make sure that no match is permitted that might subsequently turn out to be invalid. Previously, to avoid this expense, SIPE did not post constraints during the matching process. An invalid match will always be detected later by the system (usually between planning levels). The cost of permitting invalid matches is that the search must explore blind alleys. In addition, the system could fail to solve problems if the deductive operators were complex enough to trigger these invalid matches. In the toy domains on which SIPE was initially tested, these invalid matches rarely occurred and the simpler matching algorithm was preferred.

However, the complexity of the mobile robot domain triggered problems caused by invalid matching. The matcher was therefore rewritten to devote considerably more effort to matching formulas correctly. The matcher now posts constraints during the matching process itself. By posting constraints after the first predicate in a formula has been matched, the planner can shorten the search entailed in matching the other predicates. Several complications had to be dealt with, however. The constraint must be removed if the match of a subsequent predicate fails. One reason for not adding constraints originally was that the most concise constraint cannot be formulated until all predicates have been matched. Thus, the constraints added earlier might be large and could slow down all future matching. This was averted by implementing a check for constraint subsumption. The concise constraints are still computed and added after all predicates have been matched, but, if they subsume (or are subsumed by) any constraint already present, the subsumed constraint is removed.

8 Summary

We have addressed some of the practical issues involved in controlling a mobile robot with a domain-independent planning system. A method for implementing the interface between the planner and the low-level controllers on the robot was outlined. This approach is heuristic and requires that a new interface be designed for each controller. It defines an appropriate level for the planning performed by the highest-level planner, but makes no premature commitments to the design of lower-level modules. Although this leaves many difficult problems unsolved, it nevertheless permits us to begin using the SIPE planner while research continues on alternative approaches to the problems addressed by lower-level routines.

The redesigned deductive capability of the planner was described in some detail. This is a crucial feature in SIPE's ability to represent a larger class of problems than its predecessors. The extended deductive capability described here significantly increases the types of domains that can be encoded in the system, while retaining efficiency. The intermingling of execution and planning within the system provides more flexibility and greater reactivity.

Along with an explication of hierarchical planning in general, a problem, uncovered during this application, that applies to all hierarchical planners was described, and various solutions were presented and evaluated. A few examples illustrating the kind of detail that must be dealt with to produce an effective implementation were included. These practical issues have simply been ignored in much of the theoretical work done in planning.

Many central problems remain to be solved. Appropriate low-level controllers need to be developed. The interfaces between each controller and the planning system must be designed. In particular, the translation of sensory input into appropriate levels of description for the planner is still a major problem. While the approach discussed here is aimed at the short-term incorporation of domain-independent planners into integrated robots, additional problems must be resolved before a robust system capable of operating in an unconstrained environment can become reality.

References

[1] Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence 2*, 1971, pp. 189-208.

[2] Georgeff M., Lansky A. and Schoppers M., "Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot", Technical Note 380, SRI International Artificial Intelligence Center, Menlo Park, California, 1986.

[3] Goad, C., "Fast 3D Model-Based Vision", in *From Pixels To Predicates: Recent Advances In Computational And Robotic Vision*, Editor, A. Pentland, Ablex.

[4] Hayes-Roth, B. and Hayes-Roth, F., "A Cognitive Model of Planning", *Cognitive Science 3*, 1979, pp. 275–310.

[5] Hobbs, J.. "Granularity", *Proceedings IJCAI–85*, Los Angeles, California, 1985, pp. 432-435.

[6] Pentland, A.P., and Fischler, M.A., "A More Rational View of Logic or, Up Against the Wall, Logic Imperialists!", *AI Magazine*, Vol. 4, No. 4, pp. 15-18 (1983).

[7] Rosenschein, S., "Plan Synthesis: A Logical Perspective", *Proceedings IJCAI–81*, Vancouver, British Columbia, 1981, pp. 331-337.

[8] Rosenschein, S., "Formal Theories of Knowledge in AI and Robotics", Technical Note 362, SRI International Artificial Intelligence Center, Menlo Park, California, 1985.

[9] Sacerdoti, E., "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence 5 (2)*, 1974, pp. 115-135.

[10] Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, 1977.

[11] Stefik, M., "Planning and Metaplanning", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 272–286.

[12] Tate, A., "Generating Project Networks", *Proceedings IJCAI–77*, Cambridge, Massachusetts, 1977, pp. 888-893.

[13] Waldinger, R., "Achieving Several Goals Simultaneously", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 250–271.

[14] Wilkins, D., "Domain-independent Planning: Representation and Plan Generation", *Artificial Intelligence 22*, April 1984, pp. 269-301.

[15] Wilkins, D., "Recovering from Execution Errors in SIPE", *Computation Intelligence 1*, February 1985, pp. 33-45.