Recovering From Execution Errors in SIPE

D.E. Wilkins
Stanford Research Institute International
Menlo Park, CA 94025

Note: This paper originally appeared in Computational Intelligence, Vol. 1, No. 1, pp. 33-45, February 1985.

Abstract

In real-world domains (e.g., a mobile robot environment), things do not always proceed as planned, so it is important to develop better execution-monitoring techniques and replanning capabilities. ~~This paper describes~~ these capabilities in the SIPE planning system. The motivation behind SIPE is to place enough limitations on the representation so that planning can be done efficiently, while retaining sufficient power to still be useful. This work assumes that new information given to the execution monitor is in the form of predicates, thus avoiding the difficult problem of how to generate these predicates from information provided by sensors.

The replanning module presented here takes advantage of the rich structure of SIPE plans and is intimately connected with the planner, which can be called as a subroutine. This allows the use of SIPE's capabilities to determine efficiently how unexpected events affect the plan being executed and, in many cases, to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. SIPE is also capable of shortening the original plan when serendipitous events occur. A general set of replanning actions is presented along with a general replanning capability that has been implemented by using these actions.

1 Introduction

A principal goal of our research in planning and plan execution is the development of a domain-independent, heuristic system that can plan an activity and then monitor the execution of that plan. Over the last two years we have designed and implemented such a system, called SIPE (System for Interactive Planning and Execution Monitoring).[1] The basic approach to planning is to work within the hierarchical-planning paradigm, representing plans in procedural networks – as has been done in NOAH [6] and other systems. Several extensions of previous planning systems have been implemented, including the development of a perspicuous formalism for describing operators and objects, the use of constraints for the partial description of objects, the creation of mechanisms that permit concurrent exploration of alternative plans, the incorporation of heuristics for reasoning about resources, and the creation of mechanisms that make it possible to perform deductions.

Given a description of the world and a set of operators that it can apply, SIPE can generate a plan to achieve a goal in the given world. (Operators are the system's description of actions that it may perform.) However, in real-world domains, things do not always proceed as planned. Therefore, it is desirable to develop better execution-monitoring techniques and better capabilities to replan when things do not go as expected. In complex domains it becomes increasingly important to use as much as possible of the old plan, rather than to start all over when things go wrong.

This paper describes the execution-monitoring and replanning abilities that have recently been incorporated into the SIPE system. The particular advantages that can be obtained by using the rich structure in our plan representation are shown, as well as more general problems. The environment of a mobile robot has been used as a motivating domain in the development of some of the abilities here, though the implementation has been carried out in a general, domain-independent manner. This document does not describe resources, constraints, plan generation, and other features of SIPE, nor does it attempt to justify the basic assumptions underlying the system. The interested reader is referred to [10] for this.

The problem we are addressing is the following: given a plan, a world description, and some appropriate description of an unanticipated situation that occurs during execution of the plan, our task is to transform the plan, retaining as much of the old plan as is reasonable, into one that will still accomplish the original goal from the current situation. This process can be divided into four steps: (1) discovering or inputing information about the current situation; (2) determining the problems this causes in the plan, if any, (similarly, determining shortcuts that could be taken in the plan after unexpected but helpful events); (3) creating "fixes" that change the old plan, possibly by deleting part of it and inserting some newly created subplan; and (4) determining whether any changes effected by such fixes will conflict with remaining parts of the old plan. Steps 2 and 4, and possibly 3 as well, involve determining which aspects of a situation later parts of the plan depend upon. Part of this problem is an instance of the standard truth maintenance problem, and SIPE's solution is described in Section 4. In SIPE, Step 4 becomes part of Step 3, as only those fixes that are guaranteed to work are produced. In addition, serendipitous effects are used to shorten the original plan in certain cases.

The major contributions of the replanning module in SIPE result from taking advantage of the system's rich plan representation and from imbedding it within the planning system itself, rather than implementing it as an independent module. This provides a number of benefits, of which the most important follow: (1) the replanning module can exploit the efficient frame reasoning mechanisms in SIPE to discover problems and potential fixes quickly; (2) the deductive capabilities of SIPE are used to provide a reasonable solution to the truth maintenance problem described above; and (3) the planner can be called as a subroutine to solve problems after the replanning module has inserted new goals into the plan.

Another important contribution is the development of a general replanning capability (see Section 6) that has been implemented by using a general set of replanning actions. In general, recovery from an arbitrary error poses a difficult problem. Often very little of the existing plan can be reused. One can always fall back on solving the original problem in the new situation, ignoring the plan that was being executed. The replanning part of SIPE, however,

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

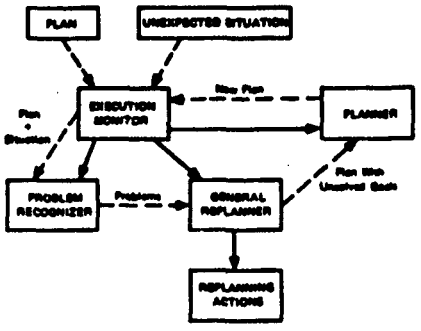


Figure 1: Control and Data Flow in SIPE Modules

tries to change the old plan, while retaining as much of it as possible. Since the problem is so difficult, one would not expect very impressive performance from a general replanner such as SIPE's.

Better performance requires domain-specific information for dealing with errors. In many domains, the types of errors that are commonly encountered can be predicted (e.g., the robot arm dropping something it was holding, or missing something it was trying to grasp). For this reason, the general replanner is based on a number of general replanning actions (i.e., actions that modify a plan in ways that are useful for handling unexpected situations) that can be referred to in a language for providing domain-specific error recovery instructions. Section 6 gives the outline of such a language.

1.1 Assumptions

SIPE assumes that information provided about unexpected events is correct and, to a certain extent, complete. This assumption avoids many of the hardest problems involved in getting a planner such as SIPE to control a mobile robot. The challenging task of determining how to generate correct predicates from information provided by the sensors is not addressed. We expect the translation of the information from the robot's sensors (e.g., the pixels from the camera or the range information from ultrasound) into the higher-level predicates used by the planner to be crucial in applying a SIPE-like planner to a mobile robot. *We hope to deal with this problem in the near future.*

In a mobile robot domain, it may often be important to expend considerable effort in checking for things that might have gone wrong besides the unexpected occurrence already noticed. There is a substantial tradeoff involved here, as interpreting the visual input of unanticipated scenes may be expensive. The research described in this paper does not examine this problem either. It assumes that nothing has gone wrong besides reported errors and effects that can be deduced from them. The problem of uncertain or unreliable sensors or information is also largely unaddressed, except that some predicates and variables may be specified as unknown. What is discussed here is what to do with new information in the form of predicates (if we assume that such predicates have somehow been discovered). Replanning appropriately with such information is an essential part of the overall solution.

1.2 Overview

Figure 1 shows the various modules in the SIPE execution-monitoring system. The solid arrows show which modules call which others. The broken arrows show the flow of data and information through the system as it replans for an unexpected situation. These arrows are labeled with a description of the data being passed.

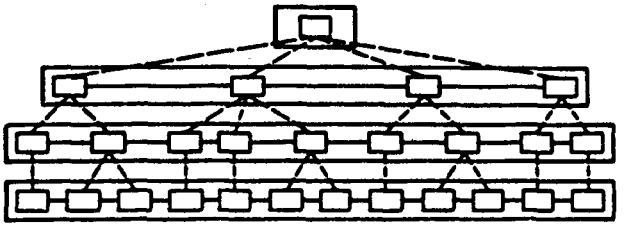
The general replanner is given the list of problems found by the problem recogniser and tries certain replanning actions in various cases, but will not always find a solution. The general replanner changes the plan so that it will look like an unsolved problem to the standard planner in SIPE (e.g., by inserting new goals). After the replanner has dealt with all the problems that were found, the planner is called on the plan (which now includes unsolved goals). If it produces a new plan, this new plan should solve correctly all the problems that were found.

Section 2 of this paper describes the features of plan representation in SIPE that are relevant to its replanning capabilities. To describe unexpected situations, a user (at present a human, but eventually this may be a program controlling and interpreting the robot's sensors) can enter arbitrary predicates at any point in the execution or can specify certain things as unknown. Section 3 describes the details of this process. Once the description of the unexpected situation has been accumulated, the execution monitor calls a problem recogniser described in Section 4, which returns a list of all the problems it detects in the plan. The replanning actions are described in Section 5 and the general replanner in Section 6. Section 7 shows examples of the general replanner in operation.

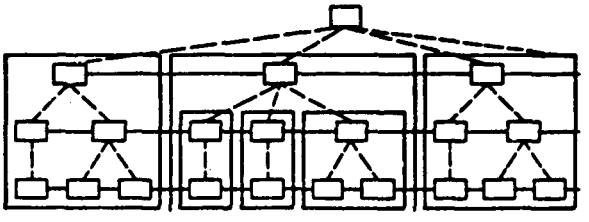
2 Plans in SIPE

Plans in SIPE are represented as procedural networks [6], with temporal information encoded in the predecessor and successor links between nodes. The plan rationale, of primary importance to the execution monitor, is encoded in the network by MAINSTEP links between nodes and by the use of PRECONDITION nodes (described below). MAINSTEP links describe how long each condition that has been achieved must be maintained. A context must also be given to specify a plan completely, as the network contains choice points from which alternative plans branch. The types of nodes that occur in plans are described below to the extent necessary for understanding the execution-monitoring capabilities.

SPLIT and JOIN nodes provide for parallel actions. SPLITs have multiple successors and JOINs have multiple predecessors so that partially ordered plans can be produced. JOIN nodes have a *parallel-postcondition* slot, which specifies the predicates that must all be true in the situation represented by the JOIN node. If a JOIN node originally has N predecessors, there will be N conjunctions of predicates that must all be true at the JOIN node. (Some



(a) Plans at Different Levels



(b) Wedges Used by the Execution Monitor

Figure 2: SIPE Plan Viewed from Different Perspectives

branches may have been linearised, so there may be fewer than N predecessors after planning.) It is easier to record this at the JOIN node (than by having *previous nodes* point to the JOIN as their purpose), since a failed parallel postcondition can more easily be retried during execution monitoring if there is easy access to all parallel postconditions. The parallel postcondition slot is filled only when the JOIN is first introduced into the plan; it is not updated as more detailed levels of the hierarchy are expanded. As long as the highest level predicates are as desired, it is assumed that the lower-level predicates are irrelevant.

COND, ENDCOND, and CONDPATTERN nodes implement conditional plans. COND and ENDCOND are similar to SPLIT and JOIN, but each successor of the COND begins with a CONDPATTERN node that determines which successor will be executed.

CHOICE nodes denote branching points in the search space. They have multiple successors, but the context selects one of these as being in the current plan. Constraints on variables may be posted relative to this choice point. Thus, if the part of a plan after a CHOICE node is removed, the corresponding choice point in the context should also be removed so that constraints that are no longer valid will be ignored.

GOAL nodes do not occur in final plans, since they represent problems that have not yet been solved. A GOAL node specifies a predicate that must be achieved, but which is not true in the situation represented by its location in the procedural network. Replanning actions will insert GOAL nodes in the plan. Each GOAL node has a MAINSTEP slot, which denotes a point later in the plan that depends on the GOAL. (This describes the rationale for having the GOAL in the plan.) Each goal must be maintained as true until the node which is its MAINSTEP is executed. A MAINSTEP slot can have the atom PURPOSE as its value, denoting that the given predicate *is* the main purpose of the plan, not preparation for some later action.

PHANTOM nodes are similar to GOAL nodes except that they are already true in the situation represented by their location in the procedural network. They are part of the plan because their truth must be monitored as the plan is being executed. They also contain MAINSTEP slots.

PROCESS nodes represent actions to be performed during execution of the plan; they also have MAINSTEP slots, as do PHANTOM and GOAL nodes. In a final plan, all PROCESS nodes will denote primitive actions. (There are also CHOICEPROCESS nodes, which are like PROCESS nodes except that they have a list of actions, one of which must be performed.)

PRECONDITION nodes provide a list of predicates that must be true in the situation represented by their location in the procedural network. Operators may specify preconditions that must obtain in the world state before the operator can be applied. The concept of precondition here differs from its counterpart in some planners, since the system will make no effort to render the precondition true. A false precondition simply means that the operator is not appropriate. Conditions that the planner should make true (and therefore backward-chain on) can be expressed as goal or process nodes.

By distinguishing between PRECONDITIONS, GOALS and PROCESSES, we effectively encode metaknowledge about how to achieve goals. SIPE will use any means to solve a goal node, only the operators listed to solve a process node, but no operators to solve a PRECONDITION node. Thus, a precondition's becoming false does not mean that it should be made into a goal; rather it means that the part of the plan produced by the operator which initially inserted this precondition is invalid. PRECONDITION nodes also help encode the rationale of a plan, since in effect they mean that the part of the plan associated with them (see below) was produced on the assumption that the predicates in the precondition were true.

In addition to the "horisontal" MAINSTEP, predecessor, and successor links within one level of a plan, there are "vertical" links between different levels of the hierarchy. Each node that is expanded by the application of an operator has descendant links to each node so produced. The descendant nodes in turn have ancestor links back to the original node one level higher in the hierarchy. Starting with a node that was expanded by an operator application, a *wedge* of the plan is determined by following all its descendant links (in the current context) repeatedly (i.e., including descendants of descendants, and so on) to the lowest level. (This definition of wedges is the same as that used by Sacerdoti [6].) Figure 2 depicts this graphically, with the large boxes in

Part (b) representing wedges. The node originally expanded by an operator application is called the *top* of the wedge. A wedge with its top at a high level in the hierarchy will generally contain many lower-level wedges within itself. The only nodes that can be the tops of wedges are GOAL, PROCESS, and CHOICEPROCESS.

Since PRECONDITION nodes are created only when an operator is applied, the part of a plan associated with a PRECONDITION node can be found by ascending along the ancestor links to the point at which the precondition first became part of the plan (once inserted, PRECONDITION nodes are copied down from level to level). The node that was expanded by an operator to create this precondition is one level higher than where the first PRECONDITION node appears and is the top of the wedge associated with each of the PRECONDITION nodes that are copied from the first one.

3 The Input of Unexpected Situations

During execution of a plan in SIPE, some person or computer system monitoring the execution can specify what actions have been performed and what changes have occurred in the domain being modeled. SIPE changes its original world model permanently so as to show the effects of actions already performed. At any point during execution, the system will accept two types of information about the domain: (1) an arbitrary predicate whose arguments are ground instances, that is now true, false or unknown; and (2) a local variable name that is now unknown. SIPE first checks whether the truth-values for the new predicates differ from its expectations and, if they do, it applies its deductive operators to deduce more changed predicates.

It is important to note that the inputing of predicates does not solve the "pixels to predicates" problem, which is the crucial issue in using a planner such as SIPE to control the actions of a robot. This problem involves translating the input of the robot's sensors (e.g., the pixels from the camera or the range information from ultrasound) into the higher-level predicates used by the planner. The research described here is concerned with what must be done with the predicates once they have been established but does not take up the question of how to determine them automatically. We hope to address this latter task in the near future.

3.1 Unknowns

Unknowns are a new addition to SIPE, as it previously assumed complete knowledge of the world. Having unknown quantities constitutes a fundamental modification because even the method of determining whether a predicate is true must be changed. If the truth-values of critical predicates are unknown, the planner will quickly fail, since none of the operators will be applicable. (Neither a negated nor an unnegated predicate in a precondition will match an unknown one.) Operators can require predicates to be unknown as part of their precondition in case there are appropriate actions to take when things are uncertain. Conditional plans have also been implemented as part of the execution-monitoring package in SIPE; thus, an operator might produce a plan with an action to perceive the unknown value, followed by a conditional plan that specifies the correct course of action for each possible outcome of the perception action. The deductive capabilities have also been enhanced so that operators can deduce that something is unknown.

The ability to specify variables as unknown is simply a tool provided by the system that will presumably be useful in some domains, particularly in a mobile robot domain. The idea behind this tool is that the location of an object may become unknown during execution. Rather than make predicates unknown, which may cause the application of operators to fail, we simply say that the variable representing the location is instantiated to the atom UNKNOWN, rather than to its original location. All predicates with this variable as an argument may then still match as if they were true. Thus, the system can continue planning as if the location were known. The only restriction is that no action can be executed that uses an unknown variable as an argument. When such an action is to be executed (e.g., go to LOCATION1), then the actual instantiation of the variable must be determined before the action is executed (possibly through a perception action). Note that it would be incorrect to continue planning if the truth-values of important predicates depended on the instantiation of the location variable. It is the responsibility of the user not to use the unknown variable if predicates depend on the latter's value.

3.2 Interpreting the input

The user need not report all predicates that have changed since many of these may be deduced by SIPE's deductive operators. The system's deductive power has been increased recently (see next section) so many effects can be deduced from certain critical predicates. SIPE does not check for additional unexpected predicates. Alternatively, we could decide on some basis (which would have to be provided as part of the domain-specific description) just how much effort to expend on perception actions to find out whether more than the minimum has gone wrong. For example, if we are told that (ON A B) is not true when we expected it to be, we might want to check to see if B is where we thought it was. As it is, SIPE will simply deduce that B is clear (if no other block is on B) and will not try to execute actions to make further checks with regard to the world. This latter procedure could be very expensive for a mobile robot in the absence of good domain-specific knowledge about what was worth checking.

There is a problem with unexpected effects in deciding how they interact with the effects of the action that was currently being executed (e.g., did they happen before, during, or after the expected effects?). Our solution to this problem is to assume that the action took place as expected and to simply insert a "Mother Nature" action after it that is presumed to bring about the unexpected effects (and things deduced from them). The system assumes that any effects of the action being executed that did not actually become true are either provided or can be deduced from the information provided. This solution interfaces cleanly and elegantly with the rest of the planner and avoids having to model the way in which the unexpected effects might interact with their expected counterparts.

4 Finding Problems in a Plan

Having just inserted a MOTHER-NATURE node (MN node) in a plan being executed, SIPE must now determine how the effects of this node influence the remainder of the plan. There are two aspects to this: the first involves planning decisions that were based on the effects of this node, and the second involves deductions about the state of the world that were based on those effects. Section 4.1 describes the problem recogniser in SIPE, which finds all problems in the remainder of the plan that might be caused by the effects of the MN node. Because of the rich information content in the plan representation (including the plan rationale), there are only six problems that must be checked. As shown in Figure 1, the problems found by the problem recogniser are given to the general replanner. The problem recogniser also notices possible serendipitous effects.

The second aspect mentioned above involves solving the traditional truth maintenance problem. Many effects deduced later in the plan may no longer be true if they depended on predicates that are negated by the MN node. The validity of such deductions must be checked so that the remainder of the plan represents the state of the world accurately. Section 4.2 describes how SIPE solves this problem, correctly updating deductions later in the plan. Deductions that are changed may or may not cause problems that should be recognised by the problem recogniser. If such problems are generated, they will be found by the problem recogniser described in Section 4.1, since the deductions are correctly updated before the problem recogniser is called.

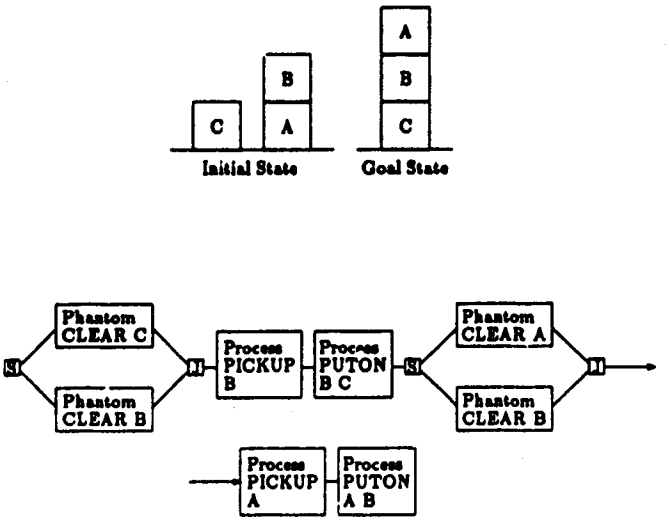


Figure 3: Blocks World Problem and Plan

4.1 Problems found by the problem recognizer

All occurrences of the six problems listed below are found by the problem recogniser. These problems constitute the only things that can go wrong with a plan in SIPE after addition of a MN node at the current execution point. The blocks-world problem in Figure 3 will be used to show an example of each type of problem.

1 - *Purpose not achieved.* If the MN node negates any of the main effects of the action just executed, there is a problem. The main effects must be reachieved. If during execution of the first PUTON node in the plan in Figure 3, either ¬(ON B C) or (ON B D) is given as an unexpected effect, then the MN node inserted after the PUTON node will negate the purpose of the PUTON node – thereby resulting in an instance of this type of problem.

2 - *Previous phantoms not maintained.* SIPE keeps a list of phantom nodes that occur before the current execution point (including those on parallel branches), and whose MAINSTEP slot specifies a point in the plan that has not yet been executed. These are phantoms that must be maintained. If the MN node negates any of these, then there is a problem. The phantoms that are no longer true must be reachieved. Suppose that during execution of the first PICKUP node in the plan in Figure 3, ¬(CLEAR C) is given as an unexpected effect. This type of problem will then occur, since the phantom node (CLEAR C) has a MAINSTEP slot (not shown in the figure) pointing to the first PUTON node, but has been negated by the MN node after the first PICKUP node.

3 - *Process node using unknown variable as argument.* If a variable has been declared as unknown, then the first action using it as an argument must be preceded by a perception action for determining the value of the variable (see Section 3). If the B in the plan were the instantiation of the variable BLOCK1 (instead of being given as part of the problem), and UNKNOWN BLOCK1 were entered during execution of the first PICKUP action, then this type of problem would occur with the immediately following PUTON action, since it would be applied to an UNKNOWN argument.

4 - *Future phantoms no longer true.* A phantom node after the current execution point may no longer be true. It must be changed to a GOAL node so that the planner will try to achieve it. In the sample plan, suppose that (ON D B) were given as an effect during execution of the first PUTON node. This type of problem would then occur with the last (CLEAR B) phantom node in the plan, since it would no longer be true when it is expected to be.

5 - *Future precondition no longer true.* A PRECONDITION node after the current execution point may no longer be true. In this case, we do not want to reachieve it, but rather pop up the hierarchy and perform some alternative action to achieve the goal at that level of the hierarchy. Because the sample plan contains no PRECONDITION nodes, we consider an example of this type in the travel planning domain. Suppose there is an operator for John's taking a taxi to the airport, which has a precondition that John's car is inoperative. If, during execution of the first part of the plan, SIPE is told that John's car is not broken, this type of problem will occur. In this case the reason for taking a taxi to the airport has been invalidated, and the general replanner will pop up the hierarchy and apply a different operator to get John to the airport (presumably driving his car).

6 - *Parallel postcondition not true.* All the parallel postconditions may no longer be true at a JOIN node. (This could be handled by maintaining phantoms, but is more convenient to handle separately.) In this case, we must insert a set of parallel goals after the JOIN, one for each untrue parallel postcondition. The parallel postconditions of the new JOIN will be the same as those on the old JOIN. In the sample plan, the last JOIN node will have both (ON A B) and (ON B C) as parallel postconditions (since they were in parallel originally). Suppose that (ON B TABLE) were given as an effect during the execution of the last PUTON node in the plan. This type of problem would then occur, since the parallel postcondition of (ON B C) would no longer be true.

Because of the way plans are encoded in SIPE, these are the only things that need to be checked when determining whether an MN node affects the remainder of a plan. This illustrates how the rich structure of plans in SIPE helps produce efficient problem detection. It should be noted, however, that processes (actions) are assumed to work whenever their precondition is true and when all phantoms whose MAINSTEP slot points to the process are true. (All such necessary conditions should be encoded as either preconditions or goals, in any case.) There is currently no check for loops caused by the same error happening repeatedly, with the same fix being proposed by the general replanner each time. Various simple checks could easily be added if this were a problem.

Finally, there are two important points to note with regard to the problem recogniser. First, in addition to the above problems, possible serendipitous effects are also noted and included in the list of problems. If the main effect of some action later in the plan is true before the action is executed, then that is noted as a possible place to shorten the plan (this is discussed in more detail in the next section). Second, only the last three problems above interact with the solution to the truth maintenance problem, since only they involve the truth-value of predicates in situations after the current execution point. The problem recogniser takes into account any changed deductions (see Section 4.2) while looking for the latter three problems.

4.2 Solution to the truth maintenance problem

SIPE's solution to the truth maintenance problem is based on the efficiency of its deductive capability. Since it is assumed that processes work as expected whenever their precondition is true and all phantoms whose MAINSTEP slot points to the process are true, only deduced effects need to be checked for their dependence on unexpected effects. (The execution monitor will solve problems having to do with preconditions and phantoms that are not true).

SIPE's deductive capability was designed to find a good balance between expressiveness and efficiency. While providing the power of many useful deductions, it nevertheless keeps deduction under control by severely restricting the deductions that can be made, as well as by having triggers to control the application of deductive operators. All deductions that can be made are performed at the time a node is inserted into the plan. Since deduction is not expensive, the truth maintenance problem is solved simply by redoing the deductions at each node in the plan after an MN node. Even this can be avoided in simple cases, because SIPE carries a list of changed predicates as it goes through the plan and, if they all become true later in the plan (without any deduced effects changing in the interim), then the execution monitor need not look at the remainder of the plan (either for redoing deductions or for finding problems).

5 Replanning Actions

The eight replanning actions described below, itREINSTANTIATE, INSERT, INSERT-CONDITIONAL, RETRY, REDO, INSERT-PARALLEL, POP-REDO, and POP-REMOVE have all been implemented in SIPE. These actions provide sufficient power to alter plans in a way that often retains much of the original plan. These are domain-independent actions, and they form the basis of the general replanner. They should also prove useful as a basis for domain-specific error recovery operators. Both of these uses are described in more detail in Section 6. The first seven actions can all be used to solve problems found by the problem recogniser, while the last is used to take full advantage of serendipitous effects.

Four of the replanning actions change the plan so that it will contain unsolved problems. The intention (see Figure 1) is that the plan will then later be given to the normal planning module of SIPE (possibly after a number of these replanning actions have changed the plan). The planner will then attempt to find a solution that solves all the problems that have been corrected in the plan. The planner automatically checks to determine whether nodes it splices into the middle of the plan cause problems later, so that any solution found will be correct. (It does this when copying nodes down to the next lower level during planning.) In all actions described below, the context argument merely specifies the context of the current plan.

• *REINSTANTIATE (predicate node context)*
This action attempts to instantiate a variable differently so as to make the given predicate true in the situation specified by the given node. This appears to be a commonly useful replanning action. For example, it might correspond to using a different resource if something has gone wrong with the one originally employed in the plan, or deciding to return to the hopper for another screw rather than trying to find the one that has just been dropped.

An attempted reinstantiation is done by looping through the arguments of the given predicate. For each argument that is a planning variable (as opposed to an actual ground instance), SIPE checks to see if there is another instantiation for it that will make the predicate true. This is cheap and efficient in SIPE, since it merely involves removing the INSTAN constraint on the variable from the current context (and also from all variables constrained to be the same as this one), and then calling the normal matcher (which will return possible instantiations) to determine if the predicate is now true. Note that all other constraints that have been accumulated on this variable are left intact, so only instantiations that meet all relevant requirements are found.

If new instantiations are found, the REINSTANTIATE action checks the remainder of the plan to see if any parts of it might be affected by the new instantiation. This is done by a routine similar to the problem detector described in Section 4 (in fact, the two share much of their code). REINSTANTIATE currently accepts new instantiations only if they cause no new problems (see discussion below on trade-offs). If all new instantiations are rejected, the old INSTAN constraint is simply replaced. Note that replanning may be done later in the plan after the REINSTANTIATE action because of other problems that were found; the only requirement here is that the REINSTANTIATE action itself not introduce new problems later in the plan.

One might use REINSTANTIATE to help with the above mentioned problem of dropping a screw in the following way. Suppose that SCREW1 is a planning variable, while S1 and S2 are particular screws. The plan being executed could have SCREW1 instantiated to S1, a phantom to be maintained with the goal of (KNOWN-LOCATION SCREW1), and a PROCESS node for moving SCREW1 to achieve (AT SCREW1 WORKBENCH). During execution of the latter node, SIPE is told that the finger separation of the arm is zero. From this it could deduce (among other things) ¬(KNOWN-LOCATION SCREW1) and ¬(AT SCREW1 WORKBENCH). The problem of not achieving the purpose of the PROCESS node will result in an (AT SCREW1 WORKBENCH) goal being inserted in the plan. Without REINSTANTIATE, this would involve finding the location of S1 and moving it to the workbench - which may be a very hard problem (as anyone who has ever dropped a screw is aware). The problem of not maintaining the phantom node could trigger REINSTANTIATE on the predicate (KNOWN-LOCATION SCREW1), which would result in SCREW1 being reinstantiated to S2 (whose location is known). This would introduce no new problems and SIPE could proceed to solve the (AT SCREW1 WORKBENCH) goal by getting S2 from the hopper.

To prevent the introduction of a large search space, REINSTANTIATE is limited by the requirement that it not introduce new problems. There are also trade-offs in deciding when to apply REINSTANTIATE as it exists, but these are discussed later in the paper. The implementation described above opts for reinstantiation only when it is likely to be the correct solution. This is consistent with SIPE's running efficiently on the problems it does solve. Alternatively, new instantiations could be accepted even though they caused problems - as long as the latter are less severe than the problems incurred by keeping the old instantiation. Since SIPE has no way of comparing the difficulty of two sets of problems, REINSTANTIATE does not do this. Doing so would introduce another very large search space into the replanning process. However, it would not be difficult to change SIPE to explore this search space if a domain warranted it. There are also ways to partially lift this restriction at the cost of a moderately increased search space (though the tradeoffs involved probably depend on the domain).

One could also expend more effort in finding new instantiations. As implemented, this replanning action will find reinstantiations when only one variable is changed. Some problems could be solved by reinstantiating a whole set of variables, but this would be more expensive and involve a search problem to decide which variables to include in the set. The decision to try only one variable was made because it is efficient while evidently powerful enough to be useful. If the ability to reinstantiate sets of variables appeared useful, implementing it would certainly be tractable.

• *INSERT (node1 node2)*
This action inserts the subplan beginning with node1 (which has been constructed) into the current plan after node2. All links between the new subplan and the old plan are inserted correctly. This is used as a subroutine by many of the actions below.

• *INSERT-CONDITIONAL (variable node context)*
This action is not very interesting, but complements the unknown variable feature, which may be useful. It simply inserts a conditional around the given node that tests whether the given variable is known. If it is, the given node is executed next; otherwise a failure node is executed.

• *RETRY (node)*
This replanning action is very simple. The given node is assumed to be a phantom node and it is changed to a goal node so that the planner will perceive it as unsolved.

• *REDO (predicate node context)*
This action creates a GOAL node whose goal is the given predicate. It then calls INSERT to place this new node after the given node in the plan. The planner will see the new node as an unsolved goal.

• *INSERT-PARALLEL (node predicates context)*
This action essentially does a REDO on each predicate in the list PREDICATES and puts the resulting GOAL nodes in parallel between a newly created SPLIT and JOIN. This subplan is inserted after NODE in the plan. The planner will see these new nodes as unsolved goals. This action is useful for reachieving parallel postconditions.

• *POP-REDO (node predicates context)*
This and POP-REMOVE are the most complicated of the replanning actions; it is used to remove a hierarchical wedge from the plan and replace it with a node at the lowest level. POP-REDO is used when a PRECONDITION node is no longer true and another action must be applied at a higher level. It could also be used to find higher-level goals from which to replan when there are widespread problems causing the replanning to fail (this is not currently implemented).

When redoing a precondition failure, it is easy to determine the wedge to be removed, since PRECONDITION nodes are copied down from one level to another. The top of the wedge to be removed is the node that was expanded to initially place the given PRECONDITION node (or one of its ancestors that is a PRECONDITION node) in the plan. Actually, only the bottom of the wedge is spliced out of the plan, as planning will continue only from the lowest level. The subplan that is removed at the lowest level is replaced by a copy of the GOAL or CHOICEPROCESS node that was at the top of the wedge. (The INSERT replanning action is used for this.) This is seen as an unsolved goal by the planner, which automatically checks to ascertain whether expansions of this node cause problems later in the plan.

Let us consider the example mentioned earlier of John planning to take a taxi to the airport when his car is broken. The operator for taking the taxi could have a precondition ¬(HAS-CAR JOHN AUTO1) ∨ (BROKEN AUTO1). (This will match John's not having a car or his car being broken.) This operator is applied to solve the GOAL node (AT JOHN AIRPORT) at a high level in the plan, causing a PRECONDITION node for the above precondition to be inserted into the plan and copied down to all lower levels of the plan. Suppose that, during execution, ¬(BROKEN AUTO1) is entered as an unexpected effect during execution of a process before the PRECONDITION node. This node is a future PRECONDITION which becomes false, and the general replanner will apply POP-REDO to the problem. The wedge that is deleted has the GOAL node (AT JOHN AIRPORT) at the top. This may be a very large wedge if its lowest level is as detailed as "find the phone book, look up taxi in the yellow pages, dial a taxi company," etc. At the lowest level, the whole plan of finding a taxi and taking it to the airport is spliced out and replaced by an (AT JOHN AIRPORT) GOAL node. When SIPE's planner is later called on this plan, this GOAL node may be solved by John's driving his car to the airport.

There is one potentially serious complication in the above description of POP-REDO. Namely, various constraints may have been posted on the planning variables because of decisions made in the wedge of the plan that has been effectively removed. Fortunately, because of SIPE's use of alternative contexts, this is easily solved. A context is a list of choice points, and constraints are posted relative to the choice point that forced them to be posted. Therefore, this problem is solved by removing from the current context all the choice points that occurred in the wedge of the plan that was effectively removed. This new context is given as the context argument to future planning actions, and no further action need be taken. This results in ignoring precisely those constraints that should be ignored.

• *POP-REMOVE (node predicates context)*
SIPE takes advantage of serendipitous effects to shorten a plan by using POP-REMOVE, which removes a wedge but does not insert a node. (It shares much of its code with POP-REDO.) However, in this case it is nontrivial to decide which wedge to remove. There may be various wedges that are candidates and, as with REINSTANTIATE, these candidates may cause problems later in the plan if they are removed. SIPE currently handles this case in the same way it handles REINSTANTIATE. Namely, it removes a wedge, checks to see if this causes any problems, and, if there are any, replaces the wedge. Thus, serendipitous effects are exploited only if doing so does not change the rest of the plan. This is a trade-off like the one discussed previously. SIPE again opts for efficiency, but could easily be changed to explore the additional search space of replanning after the removal of wedges.

SIPE also reduces the search space by generating only one candidate wedge. It gives up taking advantage of the serendipitous effect if this wedge does not work. The candidate wedge is generated by following ancestor links from the node given to POP-REMOVE (which supposedly has a purpose that has become true serendipitously), as long as some main effect of the candidate node is made true by one of the predicates in the list of given predicates (that have unexpectedly become true). The candidate node found in this manner determines the candidate wedge. The wedge is rejected immediately unless all its main effects are true in the given list of predicates.

Figure 4 uses the example of getting John to the airport to help illustrate this selection process. This example depicts a frequently occurring case in which the last action at one level of a wedge achieves the main effect of every level above that. For example, at Level 1 the goal is only to get John to the airport. At Level 2, after the choice has been made to take the taxi, the last node will achieve getting both John and the taxi to the airport. If Level 3 plans the mechanics of leaving the taxi, the last node there might contain all these higher-level effects as well as the thinner state of John's wallet.

The above selection process requires that all goals generated at a higher level and achieved in the candidate wedge be achieved before the wedge becomes a candidate, while goals generated at a lower level than the top of the candidate wedge need not have been achieved serendipitously. Thus, for Wedge 2 to be selected in Figure 4, the serendipitous effects must include (AT JOHN AIRPORT) from the higher level but need say nothing about how much cash John has since that is at a lower level. (It is assumed that, as long as the highest-level goal is achieved, we do not care about the lower-level goals that were necessary to bring this about.) The main effects of higher-level nodes that are achieved within a candidate wedge are easily checked because they are copied down as effects of the node that achieves them. Thus, checking to verify that all main effects of the candidate wedge are true ensures that all important higher-level effects will be true. In the example as shown, Wedge 2 can never be selected by this selection process because Wedge 1 will work whenever Wedge 2 does. However, in another example the effects of Wedge 1 might be achieved at Level 2 before Wedge 2 so that Wedge 2 might then be selected.

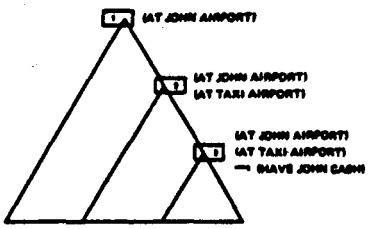


Figure 4: Hierarchical Wedges with a Common Last Action

6 Guiding the Replanning

The replanning actions of the preceding section form the basis for SIPE'S general replanning capability and for a language capable of specifying domain-specific error recovery instructions that has been designed but not implemented. The latter could be thought of as instructions for guiding the search of the general replanner. This section describes the automatic replanner and briefly outlines the error recovery operators.

6.1 The General Replanner

The general replanner takes a list of problems as well as possible serendipitous effects from the problem recogniser, and calls one or more of the replanning actions in an attempt to solve the problem. It first checks that a listed problem is still a problem, since the REINSTANTIATE action may solve many problems at once.

If the problem is a purpose that is not being achieved, the system tries a REDO, which inserts the unachieved purpose as a GOAL node after the MOTHER-NATURE node. If the problem is a previous phantom not being maintained, SIPE first tries REINSTANTIATE and, if that fails, it calls RETRY. The idea is that, if there is another object around with all the desired properties, it would be easier to use that object than to reachieve the desired state with the original object. If a PROCESS node has an unknown variable as an argument, INSERT-CONDITIONAL is called. If a future phantom is no longer true, RETRY is called. As with maintaining phantoms, REINSTANTIATE may be more appropriate, but, in both cases, this depends entirely on the domain; thus the selection here is arbitrary. For preconditions that are not true, the general replanner first calls REINSTANTIATE and, if that fails, calls POP-REDO. If parallel postconditions are not true, the general replanner calls INSERT-PARALLEL with the appropriate parallel goals.

While a general replanning capability is a significant achievement, one cannot expect very impressive performance from a replanner that does not have domain-specific information for dealing with errors. For example, whether or not REINSTANTIATE is likely to succeed will be dependent on the domain. The automatic replanner makes reasonable guesses at what might be a good choice in the domains on which SIPE has been tested. Since it merely chooses a replanning action for each type of problem that is found, it is very simple and could easily be rewritten for different domains.

6.2 Error Recovery Language

We also have designed an extension of the operator description language that enables instructions for handling foreseeable errors to be included in operators. This will allow encoding of domain-specific knowledge for guiding the search of the general replanner (or even avoiding the search altogether). The error recovery operators will have the same syntax as all other SIPE operators, with some new additions made to this language as described below. The plots of these operators will include references to the replanning actions in Section 5. SIPE's ability to specify conditional plans in operators can be used to try a second replanning action if the first one fails.

The error recovery operators will match their argument list to the arguments of the node being executed so that original problem variables can be bound to the variables in the operators. There will be two ways to invoke these operators: one for general operators that solve problems that have been recognised, and one for more specific operators that act directly on unexpected predicates. Both are described below.

The general operators will be applied after a MN node is added and problems have been found by the problem recogniser, but before the general replanner is called (see Figure 1). They will be applied to each of the problems in turn. Like deductive operators, error recovery operators will have a TRIGGER slot [10] to determine when they should be applied. The trigger will be a a combination of keywords and predicates, with the keywords referring to the six types of problems. These triggers will match when their keyword matches the problem being tried and any predicate in the trigger matches the appropriate predicates given in the problem. These operators may also have normal preconditions, which will be matched (in the normal manner) in the situation specified by the MN node. If any general error-recovery operator matches a given problem (i.e., both the trigger and precondition match), then the general replanner simply uses the instructions in the plot of the operator to choose the replanning actions to perform (rather than applying its own actions). Thus domain-specific guidance is supplied to the general replanner. It would be easy for such an operator, for example, to always force or prevent a REINSTANTIATE for a certain type of error under certain conditions.

Specific error-recovery operators are applied directly to predicates as they are inputted to the execution monitor before the problem recogniser is called. This should not be expensive because only operators mentioned in the ERROR slot of the action being executed are tried (see below). This ability seems attractive, since it can save a lot of effort when there is good domain-dependent error-recovery information available. When a specific operator matches an unexpected predicate, it may be possible in certain domains to simply apply the operator and assume that it will solve any problems caused by the unexpected predicate, thus circumventing the normal problem detection mechanism. If this option is chosen, SIPE simply assumes these error recovery operators are correct. Normal operation would involve checking for problems as usual after the application of specific operators.

Nodes in plots of regular operators will be able to specify an ERROR slot that gives names of specific error-recovery operators. When a node with an ERROR slot is being executed, the execution monitor will apply the operators listed in the error slot immediately to any unexpected predicate that is inputted during execution. Matching will be the same as for general operators, except that there are no keywords in the trigger. If one of these operators matches, the replanning actions in its plot will be carried out immediately. There may or may not be an option to preclude further problem detection on a predicate that has been so matched.

As an example of the intended use of a specific error recovery operator, consider the problem of dropping a screw that REINSTANTIATE solved in the general replanner. Suppose it is always desirable to return to the hopper after dropping a screw during the process of transporting it. The node for moving OBJECT1 to LOCATION1 would have an error slot that listed the DROP-SCREW operator. This specific error-recovery operator would match whenever the hand suddenly becomes empty and OBJECT1 is a screw. The plot of this operator could then specify a REINSTANTIATE of the variable OBJECT1 (which will still be constrained to be a screw) followed by an INSERT of a GOAL node to be achieved (AT OBJECT1 LOCATION1). Use of this operator could solve all problems of dropping a screw so that use of both the problem recogniser and general replanner could be avoided.

7 Examples

This section presents two simple examples of SIPE monitoring the execution of a simple plan, then replanning when things do not go as expected. SIPE has been tested on larger and more complex problems than those presented here. They involve a standard blocks world with ON and CLEAR predicates and a PUTON operator, as described in more detail elsewhere [10]. The user inputs only what is explicitly mentioned in boldface below; everything else is generated automatically by the system. This first problem was constructed to show the successful use of the REINSTANTIATE replanning action, and the second shows how the system inserts a newly created subplan during the replanning process.

Figure 5 shows the initial world state and the original problem. The problem is to get A on C in parallel with getting any blue block on any red block. In the initial world B1 and B2 are the only blue blocks (they are both on the table) and R1 and R2 are the only red blocks (R1 is on B1 and R2 is on the table and clear). Since A and C are both clear initially, SIPE quickly finds a two-action plan of putting A on C in parallel with putting B2 on R2, as shown in Figure 6.

This plan is then given to the execution monitor module of SIPE, which asks if P197 or P168 is to be executed first. The user types **P197** and the system asks for unexpected effects. In this case the user types **(ON D R2)** followed by **NIL** to show one unexpected effect, namely D has suddenly appeared on top of R2. This creates a MN node after P197 which also has the following effects deduced by the system: ¬(ON D TABLE) ∧¬(CLEAR R2). The problem recogniser is called and it finds only one problem, namely the PHANTOM node P165 in the parallel branch was being maintained but is no longer true. This is given to the general replanner which first tries REINSTANTIATE. This succeeds as the OBJECT1 variable in the PHANTOM node an be rebound to R1 without causing any new problems in the plan. The plan in Figure 7 is passed from the planning module back to the execution monitor (without showing phantom nodes and mainstep slots). P168 is then executed without any unexpected effects and the goal is achieved. Note that the original plan was retained in its entirety and that B2 was placed on R1 instead of R2, thus achieving the original goal of getting A on C and any blue block on any red block.

The second problem is the same as the first, except that the variable REDBLOCK1 is constrained not to be R1 (by specifying IS NOT R1 in the original problem). This will cause REINSTANTIATE to be attempted but fail, since R2 is the only other red block. The original plan produced by SIPE is the same and the unexpected situation input by the user is the same. The problem recogniser again passes the same problem to the general replanner. This time SIPE tries REINSTANTIATE and fails, so it calls RETRY, which causes the (CLEAR R2) phantom in Figure 6 to be made into a goal. The planner solves this by producing a plan that puts D back on the table before B1 is placed on R2. The subplan shown in Figure 8 replaces the (CLEAR R2) phantom node, P165, on the parallel branch before the PUTON B2 R2 node in Figure 6. Without unexpected events, the plan so constructed then executes correctly to achieve the original goal. Alternatively, more unexpected occurrences could be given during execution of the newly constructed plan and SIPE would again go through a similar loop of finding and fixing problems until the original goal is achieved.

8 Comparison with Other Systems

There is very little previous work in this area, since most domain-independent planning systems do not address the problem of replanning. Two that do [3,6] are discussed below. Tate's NONLIN [8], and Vere's DEVISER [9] do not concern themselves regarding execution. PLANX10 [1] lists "plan revision strategies" as an area for future work, but does not appear to do replanning currently. McCalla and Schneider's ELMER [5] has a module called the "executor" and claims to take an integrated view of planning and execution. The executor adds more detail to the plan by simulating execution. For example, secondary plans are added in parallel with the original plan [4] to provide a demon-like capability for handling certain situations that may arise. This is not replanning, but rather a more detailed level of planning, albeit with complex planning operations. The executor effectively produces complex, conditional plans (with possibly complex parallel interactions) for situations it forsees. It does not accept arbitrary input during execution and then replan by changing the original plan as SIPE does. In fact, the authors mention that "to allow replanning after a plan goes awry" [5] is a future step in their research.

Sussman's HACKER [7] does modify plans (as do most planners that handle parallel actions), but does not deal with unexpected occurrences. HACKER produces plans that are not correct, then simulates them to detect errors. HACKER then solves some of these errors by using a few simple actions, such as *reordering parallel actions or reachieving subgoals that have not been maintained.* Thus, *the program is actually dealing with expected, not unexpected,* occurrences. SIPE generates correct plans to begin with, then modifies them on the basis of arbitrary unexpected occurrences. What HACKER does with regard to plan modification is analogous to what the critics do in the standard planning module of SIPE. While some of the problems found by such critics are similar to those found by the problem detector in SIPE (e.g., previous phantoms not being protected), they are only a subset. SIPE also provides a richer set of replanning actions for modifying plans.

The PLANEX system at SRI International [2] was used to monitor the execution of STRIPS plans that were represented in triangle tables. PLANEX does not do replanning because it never changes the sequence of actions in the plan. However, it does allow for a weak version of the REINSTANTIATE action in SIPE where a variable can be reinstantiated and the same plan restarted. Without SIPE's ability to post constraints on variables, this is less useful. PLANEX uses the triangle table representation to determine the latest point in the plan where execution could begin in the current situation (unexpected or expected), including both the executed and unexecuted portions of the plan in this calculation. If unexpected occurrences create a situation in which restarting the plan from some point other than the current execution point would solve the problem, PLANEX would do this. (Note this may involve redoing previous actions or skipping actions that had been planned.)

Although PLANEX can restart the original plan at a different point, this should not be construed as replanning. Moreover, it is not likely to be useful in a realistic domain. The world is not so benign as to frequently have unexpected occurrences produce situations in which ones's original plan is still applicable exactly as is from some point. With very high-level examples (as in [2]), this may occasionally happen, but it will happen only rarely with detailed plans. For example, an action such as "pick up block B (wherever it may be)" can simply be repeated when B is accidentally dropped and its new location is unknown. However, if the robot must plan to go to the location of B before picking it up, the original plan will be applicable only in the unlikely event that B is accidentally dropped onto its original location.

Hayes's system [3] and Sacerdoti's NOAH [6] have addressed the replanning problem. However, the approaches used in both these systems are considerably simpler and less powerful than that of SIPE. For example, NOAH does not allow the input of arbitrary predicates, so the general replanning problem never

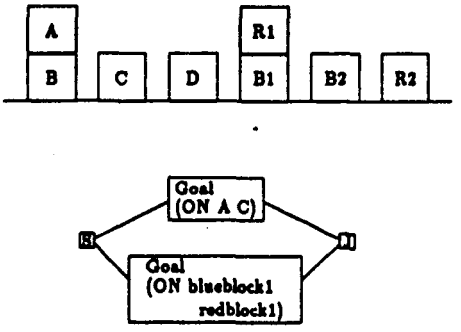


Figure 5: Initial Blocks World and Problem to be Solved

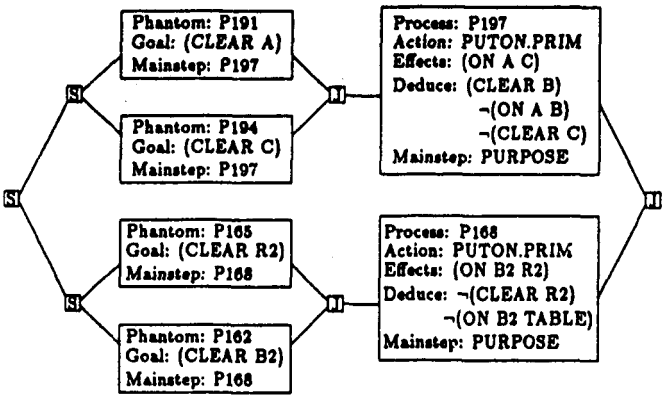


Figure 6: Initial Plan Produced by SIPE

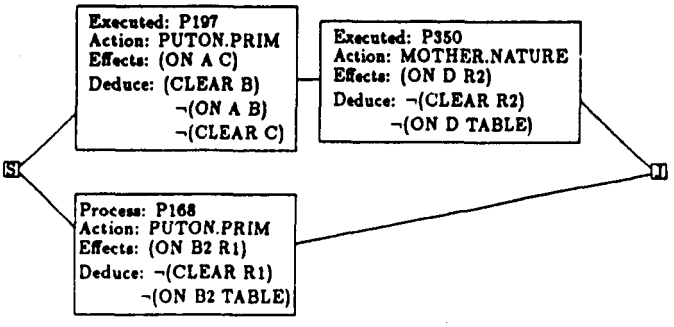


Figure 7: New Plan Produced for Continuing Execution

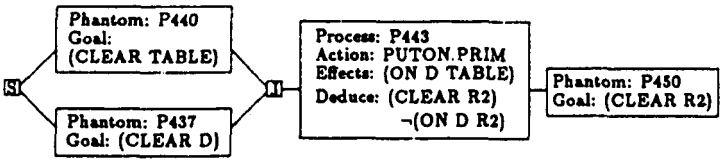


Figure 8: Subplan for Replacing PHANTOM P165

arises. It does permit the user to specify that whole wedges had been executed at once, and allows a node that has just been executed to be planned for again if it fails. This essentially provides one limited replanning action that is useful only in very specific situations.

Hayes's system does allow the input of some information about unexpected situations. It is not clear what types of information can be provided, but they appear less general than the arbitrary predicates accepted by SIPE. The system's only replanning action is to delete part of the plan. This permits the planner to reachieve higher-level goals, but they must be the same higher-level goals that were already present in the plan. The system deletes everything that depended on any effect of a decision that is no longer valid. This will in general be wasteful, since much of the plan may be removed unnecessarily. If only one of the many effects of an action has failed, subplans depending on the unfailing effects do not need to be deleted. SIPE would keep such subplans in the plan (and find any problems that may have been generated within them).

SIPE provides a much more powerful replanning capability than either of these systems. It allows the input of arbitrary predicates, computes the extent to which these affect the rest of the plan, only finds complications that are really problematical, and uses a large number of replanning actions (including REINSTANTIATE) to remedy problems in ways that enable much of the original plan to be maintained.

9 Conclusion

9.1 Summary

Given correct information about unexpected events, SIPE is able to determine how this affects the plan being executed. In many cases, it is able to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. It is also capable of shortening the original plan when serendipitous events occur. It cannot solve difficult problems involving drastic changes to the expected state of the world, but it does handle many types of small errors that may crop up frequently in a mobile robot domain. The execution-monitoring package does this without the necessity of planning in advance to check for such errors.

The major contribution of this work is the development of a general set of replanning actions that are used as the basis of an automatic replanner, as well as the basis of a language for specifying domain-dependent error recovery information. These actions provide sufficient power to alter plans in a way that often retains much of the original plan, (e.g., the REINSTANTIATE action). The general replanner attempts to solve all problems that are found. It is unlikely to be very successful unless it is adapted to particular domains. The design of the language for error recovery operators allows both for operators that will handle very specific situations and for those that will give more general advice to the replanner.

The success of these mechanisms can largely be attributed to taking advantage of the rich structure of SIPE's planner and its plans. Often, the replanner calls the standard planning system as a subroutine. In this way it can take advantage of the efficient frame-reasoning mechanisms in SIPE to discover problems and potential fixes quickly, applying its deductive capabilities to provide a reasonable solution to the truth maintenance problem. The fixes suggested by the replanner need involve only the insertion of new goals into the plan, since calling the planner as a subroutine will solve these goals in a manner that assures there will be no conflicts with the rest of the plan. SIPE's execution-monitoring capabilities make extensive use of the explicit representation of plan rationale. The problem detector makes uses of the information encoded in MAINSTEP slots, phantoms, and preconditions to quickly find all the problems with a plan. Furthermore, it does not remove parts of the original plan unless the parts are actually problematical. The replanning actions make use of constraints, alternative contexts, and wedges in SIPE whenever they consider removing part of the plan.

9.2 Issues and Limitations

From the beginning, the rationale behind SIPE has been to place enough limitations on the representation so that planning can be done efficiently, while retaining enough power to still be useful. This motivation underlies most of the design decisions that have been made in implementing the system, including the design of the replanning module. For example, REINSTANTIATE and POP-REMOVE are limited to prevent the exploration of large search spaces. The use of SIPE's deductive capability to solve the truth maintenance problem also reflects our commitment to this design philosophy. The replanning capabilities have proved useful in two test domains.

The major limitations of this research stem from the assumption of correct information about unexpected events. This avoids many difficult problems, the most important of which is generating the high-level predicates used by SIPE from information provided by the sensors. This appears to be the most critical issue in getting a high-level planner such as SIPE to control a mobile robot. Part of the problem is heuristic adequacy – the robot cannot wait ten minutes for a vision module to turn pixels into predicates while the world is changing. Other questions that have not been discussed here are deciding how much effort to expend checking facts that may be suspect, and modeling uncertain or unreliable sensors. Finding solutions to these problems is of crucial importance to the task of endowing a mobile robot with execution-monitoring capabilities.

Acknowledgments

Many people influenced the ideas expressed in this paper. Special thanks go to Michael Georgeff for many enlightening discussions.

References

[1] Bresina, J., "An Interactive Planner that Creates a Structured, Annotated Trace of its Operation", Tech. Report CBM-TR-123, Department of Computer Science, Rutgers University, December 1981.

[2] Fikes, R., Hart, P., and Nilsson, N., "Learning and Executing Generalised Robot Plans", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 231–249.

[3] Hayes, Philip J., "A Representation for Robot Plans", *Proceedings IJCAI-75*, Tbilisi, USSR, 1975, pp. 181-188.

[4] McCalla, G., and Schneider, P., "The Execution of Plans in an Independent Dynamic Microworld", *Proceedings IJCAI-79*, Tokyo, Japan, 1979, pp. 553-555.

[5] McCalla, G., and Schneider, P., "Planning in a Dynamic Microworld", *Proceedings CSCSI Conference*, Saskatoon, Saskatchewan, 1982, pp. 248-255.

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

[7] Sussman, G.J., *A Computer Model of Skill Acquisition*, Elsevier, North-Holland, New York, 1975.

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

[9] Vere, S., "Planning in Time: Windows and Durations for Activities and Goals", Jet Propulsion Lab, Pasadena, California, November 1981. (Editor's Note: AIAA Paper 83A43951)

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