DOMAIN-INDEPENDENT PLANNING: REPRESENTATION AND PLAN GENERATION

Technical Note No. 266R

May 5, 1983

By: David Wilkins, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

The research reported herein was supported by the Air Force Office of Scientific Research, Contract F49620-79-C-0188, SRI Project 8871.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **05 MAY 1983**	2. REPORT TYPE	3. DATES COVERED **00-05-1983 to 00-05-1983**
4. TITLE AND SUBTITLE **Domain-Independent Planning: Representation and Plan Generation**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **SRI International,333 Ravenswood Avenue,Menlo Park,CA,94025**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**40**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

ABSTRACT

A domain-independent planning program that supports both automatic and interactive generation of hierarchical, partially ordered plans is described. An improved formalism makes extensive use of *constraints* and *resources* to represent domains and actions more powerfully. The formalism also offers efficient methods for representing properties of objects that do not change over time, allows specification of the plan rationale (which includes scoping of conditions and appropriately relating different levels in the hierarchy), and provides the ability to express deductive rules for deducing the effects of actions. The implications of allowing parallel actions in a plan or problem solution are discussed, and new techniques for efficiently detecting and remedying harmful parallel interactions are presented. The most important of these techniques, reasoning about resources, is emphasized and explained. The system supports concurrent exploration of different branches in the search, making best-first search easy to implement.

1. The Planning Problem

The problem of generating a sequence of actions to accomplish a goal is referred to as *planning.* The automation of planning in a computer program involves representing the world, representing actions and their effects on the world, reasoning about the effects of sequences of such actions, reasoning about the interaction of actions that are taking place concurrently, and controlling the search so that plans can be found with reasonable efficiency.

The ability to reason about actions is a core problem for artificial intelligence. It is part of the common-sense reasoning people do all the time. By reasoning about actions, a program could plan your travel for you, control a robot arm in a changing environment, get a computer system or network to accomplish your computing goals, or do any number of other things that are beyond the scope of current systems. This problem is even central to conversing in natural language. First, because many utterances are deliberately *planned* to achieve specific goals, and also because it is often necessary to create your own model of other people's plans in order to understand their utterances. Despite the importance of the planning problem, relatively little has been accomplished in recent years. This paper describes progress on this problem that has resulted from development of a planning program at SRI International.[1]

Planners designed to work efficiently in a single problem domain, though desirable, often depend on the structure of that domain to such an extent that the underlying ideas cannot be readily used in other domains. Usually a new program (often requiring its own unique representation and heuristics) must be developed for each new domain to get reasonable performance. On the other hand, domain-independent planners yield planning techniques that are applicable in many domains and provide a general planning capability. Such a commonsense planning capability is likely to require different techniques from those used by an expert planning in his particular domain of expertise, but it is nonetheless essential for people in their daily lives and for intelligent programs. Of course, a general planner should provide representations and methods for including domain-specific knowledge and heuristics.

There is no guarantee, except for human performance, that a large central core of domain-independent planning techniques exists. It is important to enlarge this core as much as possible so we do not have to write a new planner for each new domain. Such motivation lies behind other AI research; for example EMYCIN [14] is an attempt to clarify the domain-independent core of expert systems such as MYCIN, and TEAM [2] provides natural-language access to data bases

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

independent of the domain or structure of the data base. This paper describes an implemented planning program that expands the core of domain-independent planning techniques as it builds on and extends such previous domain-independent planning systems as Sacerdoti's NOAH [9], Tate's NONLIN [13], Sridharan's PLANX10 [10], Vere's DEVISER [15], and SRI's STRIPS [1].

Two features found in many planning systems are also central in this work: hierarchical planning and parallel actions. Hierarchical planning is often necessary for real-world domains, since it helps avoid the tyranny of detail that would result from planning at the most primitive level. The planner can significantly reduce the search space by first planning at abstract levels and then expanding these abstract plans into more detailed plans. Parallel actions are also useful for real-world domains. Such domains are often multieffector or multiagent (e.g., having two robot arms to construct an object, or two editors to work on your report), and the best plans should use these agents in parallel whenever possible. This distinguishes planning from much of the work in program synthesis, since the goal there is often a strictly sequential program. A planning system that allows parallel actions must be able to reason about how actions interact with one another, since interference between parallel actions may prevent the plan from accomplishing its goal. This is a major problem for planning systems and a primary focus of this paper.

2. Overview of SIPE

We have designed and implemented (in INTERLISP) a system, SIPE (System for Interactive Planning and Execution Monitoring), that supports domain-independent planning. The program has produced correct parallel plans for problems in four different domains (the blocks world, cooking, aircraft operations, and a simple robotics assembly task). The system allows for hierarchical planning and parallel actions. Development of the basic planning system has led to several extensions of previous systems. These include the development of a perspicuous formalism for encoding descriptions of actions, the use of constraints to partially describe objects, the creation of mechanisms that permit concurrent exploration of alternative plans, the incorporation of heuristics for reasoning about resources, mechanisms that make it possible to perform simple deductions, and advanced abilities to reason about the interaction among parallel actions.

SIPE can generate plans automatically, but, unlike its predecessors, SIPE is designed to also allow interaction with users throughout the planning and plan execution processes. The user is able to watch and, when he wishes, guide and/or control the planning process. Our concern with interaction means that perspicuous representations have been favored and that some search control issues have not been addressed as yet. As the system evolves, more methods for controlling the search will be developed and more problems will be solved automatically. This evolutionary approach has several advantages. From the planning point of view it allows us to address larger, "real-world" problems that may initially be beyond the capabilities of fully automatic planning techniques, but which could provide interesting research problems. Development of an interactive planner also encourages us to deal with the issue of representing the planning problem in terms that can be easily communicated to a user. This is also important for an automatic planner, because the machine must still be able to communicate about the planning it has performed. Our system raises issues in human-machine interaction, but this paper addresses only the planning aspects of our work.

In SIPE a plan is a set of partially ordered goals and actions, which is composed by the system from operators (the system's description of actions that it may perform). By simply applying operators, plans that do not achieve the desired goal may sometimes be generated, so the system has *critics* that find potential problems and attempt to avert them. In particular, most of the reasoning about interactions between parallel actions is done by the critics. The plans are represented in procedural nets [9], primarily for graceful interaction between man and machine. Invariant properties of objects in the domain are represented in a sort hierarchy, which allows inheritance of properties and the posting of constraints on the values of attributes of these objects. The relationships that change over time – and therefore all goals – are represented in a version of first-order predicate calculus that is typed and interacts with the knowledge in the sort hierarchy. Operators are represented in an easily understood formalism [6] in which the ability to post constraints on variables is a primary feature. Each of these parts of the system will be described in more detail later.

It should be noted that, like most domain-independent planning systems (DEVISER being an exception), ours assumes discrete time, discrete states, and discrete operators. Time need not be represented explicitly for many tasks, since the ordering links in the procedural network provide the necessary ordering information. It is assumed that each world state that can be reached is discrete and can be represented explicitly. The operators are also discrete, with the effects of an action occurring instantaneously as far as the system is concerned. This applies to a given abstraction level; using hierarchical planning, the system can order the effects that occur at a lower level of detail. These assumptions are acceptable in many real-world domains and have been made by most previous planners. They are however restrictive and prevent many real-world phenomena from being adequately represented. For example, sophisticated reasoning about time and modeling of dynamic processes are not possible within our present framework. Few artificial intelligence programs have addressed these problems (McDermott's recent work being a notable exception [5]).

This paper describes SIPE in more detail by discussing its solutions to four major problems a planner must address. These problems are representation (of the domain, goals, and operators), recognizing and dealing with parallel interactions, controlling the search, and monitoring execution. The next four sections describe these problems and stress new developments in SIPE by comparing it to previous systems.

Most of the new developments are in the areas of representation and parallel interactions. The primary contributions of SIPE are a general constraint language and the ability to reason about resources. The use of constraints makes SIPE more powerful than previous domain-independent planners because it can use constraints both to represent a wider range of domains and to find solutions more efficiently. Resources are a powerful mechanism for reasoning about parallel actions. SIPE allows actions to use objects as resources, then reasons about these resources to find parallel actions that must be linearized. While the concept of resources in SIPE is limited, it is nevertheless quite useful both in the representation of domains and in finding solutions efficiently.

Other important representational contributions of SIPE include deductive operators, efficient methods for representing invariant properties of objects, and mechanisms for representing plan rationale. Many effects of actions are deduced (and thus do not need to be represented explicitly), but the deduction is kept under tight control – thus avoiding the search explosion encountered by more deduction-oriented planners. For greater efficiency, SIPE provides different representations (both a frame-based hierarchy and predicate calculus) for different parts of the domain. (Most domain-independent planners have used a single representation.) In addition, SIPE enables a rich description of plan rationale, thus providing important information for execution monitoring, plan explanation, and critics. These features are described in more detail below.

3. Representation

A central concern in designing a representation for a planning system is how to represent the effects an action has on the state of the world. This means that the frame problem [9] must be solved in an efficient manner. Since we intend that many domains will be encoded in the planning system, it is also necessary that the solution to the frame problem not be too cumbersome. For example, one does not want to have to write a large number of frame axioms for each new action that is defined.

The planning-representation problem involves representing the domain, goals, and operators. Operators are the system's representation of actions that may be performed in the domain or, in the hierarchical case, abstractions of actions that can be performed in the domain. An operator includes a description of how each action changes the state of the world. In a logical formalism such as Rosenschein's adaptation of dynamic logic to planning [8], the same representational formalism may be used for representing the domain, goals, and operators. However, in many planners more concerned with efficiency, including SIPE, there is a different representation for each. The goal is to have a rich enough representation so that many interesting domains can be represented (an advantage of logical formalisms), but this must be measured against the ability of the system to deal with its representations efficiently during the planning process.

3.1 Representation of Domain Objects and Goals

The system provides for representation of domain objects and their invariant properties by nodes linked in a hierarchy. This permits SIPE to incorporate the advantages of frame-based systems (primarily efficiency), while retaining the power of the predicate calculus for representing properties that do vary. Most previous domain-independent planners have used a single representation. Invariant properties do not change as actions planned by the system are performed (e.g., the size of a frying pan does not change when something is cooked in it). Each node can have attributes associated with it and can inherit properties from other nodes in the hierarchy. The values of attributes may be numbers, pointers to other nodes, key words the system recognizes, or any arbitrary string (which can be used only by checking whether it is equal to another such string). The attributes are an integral part of the system, since planning variables contain constraints on the values of attributes of possible instantiations. Constraints are an important part of the system and are discussed in considerable detail later. There are different node types for representing variables, objects, and classes, but these will not be discussed in detail here, since they are similar to those occurring in many representation formalisms – for example, semantic networks and UNITS [11].

All parts of the system are actually nodes in this hierarchy; for example, procedural net nodes, operators, and goals are all represented as nodes in the hierarchy. However, unlike variables and domain objects, these latter nodes are near the top of the hierarchy and make use of neither the inheritance of properties nor the constraints on attribute values. The use of a uniform formalism for all parts of the system has been helpful both in implementation and in interaction with users.

A restricted form of first-order predicate calculus is used to represent properties of domain objects and the relationships among them that may change with the performance of actions; it is also therefore used to describe goals as well as the preconditions and effects of operators. Quantifiers are allowed whenever they can be handled efficiently. Universal quantifiers are always permitted in effects, and over negated predicates in preconditions. Existential quantifiers can occur in the preconditions of operators, but not in the effects. Disjunction is not allowed. These restrictions result from using "add lists" to solve the frame problem. (Why this is so is described in the next section). By representing the invariant properties of the domain separately, SIPE reduces the number of formulas in the system and makes deductions more efficient. There is currently no provision for creating or destroying objects as actions are executed, although in some domains this would be useful (e.g., after an omelet has been made, do the original three eggs still exist as objects?).

3.2 Representation of Operators

Operators representing actions the system may perform contain information about the objects that participate in the actions (represented as resources and arguments of the actions), what the actions are attempting to achieve (their goals), the effects of the actions when they are performed, and the conditions necessary before the actions can be performed (their preconditions). Before SIPE's representation is described in detail, some basic assumptions made by SIPE about the effects of actions need to be presented.

Determining the state of the world after actions have been performed (e.g., the planner must ascertain whether the intended goals have been achieved), involves solving the frame problem. Here we make what Waldinger [16] has called the STRIPS assumption, which is that all relations mentioned in the world model remain unchanged unless an action in the plan specifies that some relation has changed. In STRIPS, an action specifies that a relation has changed by mentioning it on an "addlist" or "deletelist". SIPE is actually an extension of this idea, since it allows some of the changed relations to be deduced instead of being mentioned explicitly in the operators. However, all deduced effects are mentioned explicitly in the final plan, so the basic approach to the frame problem is the same. An alternative to making the STRIPS assumption would be to deduce all changes from general frame axioms. That approach was not taken in SIPE because the deduction problem is large and must be tightly controlled.

Making this assumption imposes requirements on the formalism used for representing the domain, since it must support the STRIPS assumption. While the STRIPS assumption may be very limiting in the representation of rich domains, such as automatic programming, there are many domains of interest for which it causes no problems. For example, the fairly simple environments in which robot arms often operate appear adequately representable in a system embodying the STRIPS assumption. SIPE currently makes the closed-world assumption: any negated predicate is true unless the unnegated form of the predicate is explicitly given in the model or in the effects of an action that has been performed. This is not critical; the system could be changed to assume that a predicate's truth-value is unknown unless an explicit mention of the predicate is found in either negated or unnegated form. (Although in large domains, there may be an enormous number of predicates that are not true.) Deduction in SIPE does not violate the closed-world assumption; it is used only to deduce effects of an action when the action is added to a plan (thus sparing the operator that represents the action from having to specify these effects).

Many features combine to make SIPE's operator description language an improvement over operator descriptions in previous systems. These features will be presented by discussing the sample operator given in Figure 1, with subsections devoted to the more important features. SIPE has produced correct parallel plans for problems in four different domains, one of which is the blocks world (described in [9]) for which many domain-independent planning systems (e.g., NONLIN and NOAH) have presented solutions. To facilitate comparison with these systems, a PUTON operator for the blocks world in the SIPE formalism is shown in Figure 1.

3.2.1 Interfacing Different Levels of Description

The operator's effects, preconditions, purpose, and goal are all encoded as first-order predicates on variables and objects in the domain. (In this case, BLOCK1 and OBJECT1 are variables.) Negated predicates that occur in the effects of an operator essentially remove from the model a fact that was true before, but is no longer true. Although the operator in Figure 1 has no precondition, 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 make the precondition true. A false precondition simply means that the operator is not appropriate. Conditions that the planner should make true (which may be referred to as preconditions in other planners) are expressed as goals (e.g., the CLEARTOP goals in Figure 1, which are described below).

Separating this idea of precondition from the goals gives SIPE greater flexibility for two reasons. First, it is plausible that some domains may have actions that will work in certain (possibly undesirable) situations, but that one would not want to work to achieve such a situation for the sake of performing that action. SIPE can easily represent this, whereas a planner that tried to

```
OPERATOR: PUTON
ARGUMENTS: BLOCK1, OBJECT1 IS NOT BLOCK1;
PURPOSE: (ON BLOCK1 OBJECT1);
PLOT:
    PARALLEL
        BRANCH 1:
            GOALS: (CLEARTOP OBJECT1);
            ARGUMENTS: OBJECT1;
        BRANCH 2:
            GOALS: (CLEARTOP BLOCK1);
            ARGUMENTS: BLOCK1;
    END PARALLEL

    PROCESS
    ACTION: PUTON.DETAILED;
    ARGUMENTS: OBJECT1;
    RESOURCES: BLOCK1;
    EFFECTS: (ON BLOCK1 OBJECT1);
END
```

Figure 1
A PUTON Operator in SIPE

achieve all preconditions might try to make the situation worse in order to apply its "emergency" operators. Second, SIPE's preconditions are useful for connecting different levels of detail in the hierarchy. The precondition of an operator might specify that certain higher-level conditions must be true, while the operator itself specifies goals at a more detailed level. This provides an interface between two different levels of description that was not present in NOAH. (While NOAH's plans were hierarchical, they used the same level of description at all levels of the hierarchy). In addition, SIPE's preconditions are important for representing deductive operators as described in Section 3.6.

For example, consider the PUTON.DETAILED operator in Figure 1. The reason for planning at this most abstract level with PUTON (and a CLEAR operator that is not shown) is to produce a series of PUTON.DETAILED operators that will move the blocks in the correct order. The PUTON.DETAILED operator would then plan the next level of detail, perhaps planning approach vectors to the blocks and finger-moving actions so that commands for a robot arm could be generated. The precondition for PUTON.DETAILED could specify (CLEAR BLOCK1) and (CLEAR OBJECT1). These goals should have been achieved at a higher level; if they have not, one would not want to plan them at this level of detail. Rather one should go up the hierarchy and plan them at the proper level. With proper operators in a perfect world where planning always begins at the highest level, this would be a redundant check (though still useful for explanatory purposes). In an imperfect world, this technique can be useful for replanning during execution monitoring and for detecting inappropriate commands given by the user interactively (e.g., to plan an action down to a lower level before the proper higher-level goals have been achieved).

The ability to interface between levels is further enhanced by allowing operators to specify the "purpose" they are trying to achieve at both levels of detail. The PURPOSE attribute of the operator specifies what it is trying to achieve at the lower level of detail and is used to determine plan rationale as described in Section 3.5. The GOAL attribute of an operator specifies what the operator is trying to achieve at the higher level of detail. This is used to determine when to apply the operator (i.e., what goals it will solve). For example, the PUTON.DETAILED operator might have (ON BLOCK1 OBJECT1) as its GOAL, but might also have a PURPOSE that mentions actual coordinates planned about at the lower level of detail. The operator in Figure 1 does not have a GOAL attribute (in which case SIPE's default is to use the PURPOSE attribute to serve the function of both the PURPOSE and GOAL attributes). A GOAL attribute is not needed here, since both levels of description are the same (i.e., blocks with ON and CLEARTOP predicates). The ability of the formalism to interface two levels of description is important because the real advantage of hierarchical planning stems from the ability to employ different levels of abstraction.

3.2.2 Plots

Operators contain a plot that specifies how the action is to be performed in terms of actions and goals. Like plans, plots are represented as procedural networks. When used by the planning system, the plot can be viewed as instructions for expanding a node in the procedural network to the next level. The plot may specify the next level in the plan at the same level of description as the current one (e.g., in NOAH's blocks world the level of description never changes), or at a more detailed level of description. The plot of an operator can be described either in terms of *goals* to be achieved (i.e., a predicate to make true), or in terms of *processes* to be invoked (i.e., an action to be performed). (NOAH represented a process as a goal with only a single choice of action.) Encoding a step as a process implies that only the action it defines can be taken at that point, while encoding a step as a goal implies that any action can be taken that will achieve the goal. Another less explicit difference between encoding a step as a goal or as a process is whether the emphasis is on the situation to be achieved or the actual action being performed.

During planning, an operator is used to expand an already existing GOAL or PROCESS node in the procedural network to produce additional procedural network structure at the next level. For example, the PUTON operator might be applied to a GOAL node in a plan whose goal predicate is (ON A B). Operators contain lists of resources and arguments to be matched with the resources and arguments of the node being expanded. In our example, A and B in the GOAL node are matched with BLOCK1 and OBJECT1 in the PUTON operator when the operator is used to expand the node. The plot of the operator is used as a template for generating two GOAL nodes and one PROCESS node in the plan.

Operators in SIPE provide for posting of constraints on variables, specification of resources, explicit representation of the rationale behind each action, and the use of deduction to determine the effects of actions. Each feature is described below in some detail. SIPE also provides the ability to apply the plots of operators to lists of objects. Variables in an operator can be instantiated to a list of objects by calling a generator function given in the GENERATOR attribute of the CLASS node of the variable. For example, suppose BLOCKS1 is a variable in an operator and the CLASS node for BLOCKS in the sort hierarchy has a function as the value of its GENERATOR attribute. In this case, application of that operator will result in the function being called to instantiate BLOCKS1 (e.g., to a list of blocks). The plot of the operator may then contain an ITERATE-BEGIN and an ITERATE-END, and the plot within these tokens will be reproduced in the resultant procedural net – once for each block in the list BLOCKS1. For example, a CLEARTOP goal could be generated for each block in the list. This is useful for generating paths to move along in the aircraft operations domain, as described in the section on performance.

3.3 Constraints

One of SIPE's most important advances over previous domain-independent planning systems is its ability to construct partial descriptions of unspecified objects. This ability is important both for domain represention (e.g., objects with varying degrees of abstractness can be represented in the same formalism) and for finding solutions efficiently (since decisions can be delayed until partial descriptions provide more information). The subsections below describe the constraint language, explain how it is incorporated into the system, and compare it with other systems. Since almost no previous domain-independent planning systems have used this approach (e.g., NOAH cannot partially describe objects), the constraints in SIPE will be documented in some detail.

3.3.1 SIPE's Constraint Language

Planning variables that do not yet have an instantiation (these are INDEFINITE nodes in the sort hierarchy) can be partially described by setting constraints on the possible values an instantiation might take. This allows instantiation of the variable to be delayed until it is forced or until as much information as possible has been accumulated, thus preventing incorrect choices from being made. Constraints may place restrictions on the properties of an object (e.g., requiring certain attribute values for it in the sort hierarchy), and may also require that certain relationships exist between an object and other objects (e.g., predicates that must be satisfied in a certain world state). SIPE provides a general language for expressing these constraints on variable bindings so they can be encoded as part of the operator. During planning, the system also generates constraints that are based on interactions within a plan, propagates them to variables in related parts of the network, and finds variable bindings that satisfy all constraints.

The allowable constraints in SIPE on a variable V are listed below:

- CLASS. This constrains V to be in a specific class in the sort hierarchy. In SIPE's operator description language there is implicit typing based on the variable name; therefore, in the PUTON operator in Figure 1, the variable created for BLOCK1 has a CLASS constraint that requires the instantiation for the variable to be a member of the class BLOCKS. Similarly, the OBJECT1 variable has a CLASS constraint for class OBJECTS.

- NOT-CLASS. V must be instantiated so that it is not a member of a given class.

- PRED. V must be instantiated so that a given predicate (in which V is an argument of the predicate), is true. This results in an explicit number of choices for V's instantiation, since all true facts are known (by the closed-world assumption).

- NOT-PRED. V must be instantiated so that a given predicate (in which V is an argument of the predicate) is not true.

- SAME. V must be instantiated to the same object to which some other given variable is instantiated.

- NOT-SAME. V must not be instantiated to the same object to which some other given variable is instantiated. In the PUTON operator in Figure 1, the phrase "IS NOT BLOCK1" results in a NOT-SAME constraint being posted on both BLOCK1 and OBJECT1 that requires they not be instantiated to the same thing. Thus, if SIPE is looking for a place to put block A, it will not choose A as the place to put it.

- INSTAN. V must be instantiated to a given object. This could be represented by using SAME applied to objects as well as variables (or using PRED with an EQ predicate), but instantiation is a basic function of the system and warrants its own constraint for a slight gain in efficiency.
- NOT-INSTAN. V must not be instantiated to a given object.
- OPTIONAL-SAME. This is similar to SAME, but merely specifies a preference and is not binding. For example, one would prefer to conserve resources by making two variables be the same object, but, if this is not possible, then different objects are acceptable.
- OPTIONAL-NOT-SAME. This is similar to NOT-SAME, but is not binding. If SIPE notices that a conflict will occur between two parallel actions if two variables are instantiated to the same object, then it will post an OPTIONAL-NOT-SAME constraint on both variables. If it is possible to instantiate them differently, a conflict is avoided. If it is not, they may be made the same – but the system will have to resolve the ensuing conflict (perhaps by not doing things in parallel).
- Any attribute name. This requires a specific value for a specific attribute of an object. For example, the PUTON operator could have specified "BLOCK1 WITH COLOR RED". This would have created a constraint on BLOCK1 requiring the COLOR attribute (in the sort hierarchy) of any possible instantiation to have the value RED. For attributes with numerical values, "greater than" and "less than" can also be used. In planning an airline schedule, for example, the operator used for cross-country flights might contain the following variable declaration: "PLANE1 WITH RANGE GREATER THAN 3000".

3.3.2 Using Constraints

Constraints add considerably to the complexity of the planner because they interact with all parts of the system. For example, to determine if a goal predicate is true, SIPE must verify whether it matches predicates that are effects earlier in the plan. This may require matching the variables that are arguments to the two predicates, which in turn involves determining whether the constraints on the variables are compatible. In a similar way, constraints also interact with the deductive capability of the system (to be described later). Constraints also affect critics, since determining if two concurrent actions interact may depend on whether their constraints are compatible. SIPE must also solve a general constraint satisfaction problem with reasonable efficiency, although how to control the amount of processing spent on constraint satisfaction is an open and important question. SIPE currently uses a simple and straightforward constraint satisfaction algorithm, though it is modular and replaceable.

During planning, SIPE assumes that it is possible to satisfy any constraint that it cannot prove unsatisfiable. Actual checking by running the constraint satisfaction routine is done only once per level in the hierarchy by the automatic search (as currently implemented). It can also be invoked by a user interactively. This is a good strategy because SIPE discovers most unsatisfiable constraints quickly by using the PRED constraint, which is quite powerful, and by immediately propagating many consequences as soon as constraints are posted.

When SAME and NOT-SAME constraints are added to a variable, similar constraints are immediately propagated to all other variables involved. (This may result in forced instantiations.) Whenever a constraint other than these two is added to a variable, the system verifies that at least one object satisfies all the constraints on this variable. This is fast because of efficient retrieval from the sort hierarchy and because PRED constraints store explicit disjunctive lists of possible instantiations. If only one possible instantiation remains for a variable, that instantiation is made immediately; if no instantiations remain, the current search branch fails immediately.

By accumulating PRED constraints on a variable, the system is assured that any object satisfying all the constraints on a variable will have all the properties required by the plan. A failure in constraint satisfaction can occur only when chains of SAME or NOT-SAME constraints prevent compatible assignments to different variables (each of which has an acceptable instantiation modulo those constraints). For these reasons, most problems are discovered quickly; thus, only occasional checking of the entire constraint satisfaction problem yields acceptable performance.

3.3.3 Comparison with Other Systems

The use of constraints is a major advance over previous domain-independent planning systems. NOAH, for example, would have to represent every property of an object as a predicate and then, to get variables properly instantiated, would have each such predicate as either a precondition of an operator or a goal in the plan. In SIPE an operator might declare a variable as "CARGOPLANE1 WITH RANGE 3000" and the plan using this variable can assume it has the proper type of aircraft. In NOAH goals similar to (CARGOPLANE X) and (RANGE X 3000) would have to be included in the operator and achieved as part of the plan. This makes both the operators and plans much longer and harder to use and understand. In addition to syntactic sugar, constraints in SIPE improve efficiency and expressibility. The OPTIONAL-SAME and OPTIONAL-NOT-SAME constraints used in resource reasoning cannot be expressed as goals or preconditions in a system like NOAH. The constraint satisfaction algorithm used in SIPE takes advantage of the fact that invariant properties of objects are stored directly in the sort hierarchy. The lookup of such properties in

SIPE is much more efficient than the process of looking through the plan to determine which predicates are currently true, as would have to be done in systems like NOAH and NONLIN.

Some domain-dependent systems make use of constraints. Stefik's system [11], one of the few existing planning systems with the ability to construct partial descriptions of an object without identifying it, operates in the domain of molecular genetics. Our system extends Stefik's approach in three ways. (1) We provide an explicit, general set of constraints that can be used in many domains. Stefik does not present a list of allowable constraints in his system; moreover, some of them that are mentioned seem specifically related to the genetics domain. (2) Constraints on variables can be evaluated before the variables are fully instantiated. For example, a set can be created that is constrainable to be only bolts, then to be longer than one inch and shorter than two inches, then to have hex heads. This set can be used in planning before its members are identified in the domain. (3) Partial descriptions can vary with the context, thus permitting simultaneous consideration of alternative plans involving the same unidentified objects. This is described in more detail in the section on search control.

3.4 Resources

One of the major contributions of SIPE is the ability to reason about resources. The formalism for representing operators includes a means of specifying that some of the variables associated with an action or goal will actually serve as resources for that action or goal (e.g., BLOCK1 is declared as a resource in the PUTON.PRIMITIVE action of the PUTON operator in Figure 1). Resources are to be employed during a particular action and then released, just as a frying pan is used while vegetables are being sauteed in it. Reasoning about resources is a common phenomenon. It is a useful way of representing many domains (e.g., planning an airline flight schedule or planning the cooking of a meal), a natural way for humans to think about problems, and, consequently, an important aid to interaction with the system.

SIPE has specialized knowledge for handling resources; declaration of a resource associated with an action is a way of saying that one precondition of the action is that the resource be available. Mechanisms in the planning system, as they allocate and deallocate resources, automatically check for resource conflicts and ensure that these availability preconditions will be satisfied. One advantage of resources, therefore, is that they help in the axiomatization and representation of domains. The user of the planning system does not have to axiomatize the availability of resources in the domain operators as a precondition. (Such an axiomatization may be difficult, since the critics must use the representation correctly to recognize problems caused by the unavailability of resources.)

Resources enable SIPE's operators and plans to be more succinct and easier to understand than similar operators and plans in domain-independent parallel planning systems, such as NOAH and NONLIN. In the latter systems, resource availability would have to be correctly axiomatized, checked, and updated in the preconditions and effects of the operators. It is not clear that it would be possible to do this so that the critics would recognize only the intended conflicts. If it were indeed possible, the resource reasoning in SIPE would be much more efficient than such an axiomatization in the other systems. For example, in NOAH the resolve-conflicts critic would eventually have to notice that posted "available-resource" effects are in conflict. This could be done only after the entire plan had been expanded and the critics applied. Even then, conflicts between uninstantiated variables might not be detected, since only in an attempt to instantiate them would an actual conflict arise. In SIPE it is known which resources an operator needs before it is applied, so conflicts can be detected even before the plan is expanded. This can result in choosing operators that do not produce conflicts, thereby pruning the search space. SIPE avoids not only the immediate incorrect operator expansion, but also both the entire expansion to the next level and the application of the critics after that. The savings can be considerable in domains that use resources heavily. SIPE can also detect conflicts between uninstantiated variables; if a plan requires two arms as resources and only one arm exists in the world, SIPE can detect this conflict even though the two arm variables have not been instantiated.

The concept of resource implemented in SIPE is fairly limited. More details on how resources are used are given in Section 4.

3.5 Plan Rationale

SIPE provides more flexibility in specifying the rationale behind a plan than many domain-independent planners. (The rationale for an action in a plan is "why" the action is in the plan.) This is needed for determining how long a condition must be maintained, what changes in the world cause problems in the plan, and what the relationship is among different levels in the hierarchy. SIPE constructs links, both between the levels of a plan and within a level, that help express the rationale behind the actions. Both of these are discussed below.

In the procedural networks that represent plans, PROCESS and GOAL nodes represent an action to be performed or a goal to be achieved. When a node is planned to a greater level of detail by applying an operator, the expansion may consist of many nodes. To ascertain when the effects of the higher-level node become true in the more detailed expansion, it must first be established which node in the expansion achieves the higher-level goals. Each operator in SIPE has a PURPOSE attribute that specifies a conjunction of predicates, which is the main purpose of any expansion produced with this operator. When the operator is used to produce an expansion, this PURPOSE attribute is used to determine the node in the piece of procedural net produced by the expansion that achieves the operator's purpose. The higher-level effects are then copied down to this node, and the rationale for that node being in the plan is that it achieves the higher-level goal. If the operator does not have a PURPOSE attribute, the default is to copy the effects down to the last node of the expansion. In NOAH the assumption was that the last node of an expansion achieved the main purpose, so the effects were always copied down to that node. SIPE therefore provides additional flexibility - for example, operators that include some "clean-up" or normalization after accomplishing their goal can be represented correctly.

SIPE also keeps track of the rationale for each node that is not there to achieve some higher-level goal. Such a node is put in a plan for the purpose of preparing some later action at that level, and this fact must be recorded so the planner will be able to determine how long a goal condition must be maintained. Nodes within the plot of a SIPE operator may specify later nodes in the plōt as the action they are preparing, thus explicitly stating the scope of the node. If no such scope is specified, the default (which takes advantage of the above flexibility) is that the scope of all nodes in the expansion before the node that achieves the purpose of the operator runs to that "main-purpose" node.

For example, if an expansion consists of three nodes, it may be the case that the first prepares the second, which in turn prepares the third, or it may be that the first two both prepare the third (which we'll consider to be the "main purpose" of the expansion). NOAH always assumes the latter case is true, which is the default in SIPE if no scoping information is present in the operator. However, SIPE also provides the ability to represent the first case, since the first node could explicitly state that its scope was to last only until the second node. This is important for correcting problematic parallel interactions within a plan and for execution monitoring. If the goal achieved by the first node suddenly became unexpectedly false after the second node was executed (but before execution of the third node), SIPE could perceive that there is no problem, since the first node had to be maintained only until the second node. Such an ability seems desirable. For example, one might wish to represent the operator for seasoning stew as "open the cupboard", then "pick up a bay leaf", then "put bay leaf in stew". If the cupboard closes after the bay leaf is in hand but before it is put into the stew, the planner should realize that there is no problem.

An alternative approach would be to always have the scope last only until the next node. However, one might want to represent the same operator as "take the lid off the stew pot", then "pick up the bay leaf", then "put bay leaf in the stew". In this case, it is a problem if the lid is put back prematurely. Whether or not it is possible to solve these problems by using only the default method for scoping (by forcing the user to write possibly contorted operators), the freedom to write operators naturally in unforeseen domains seems to be an advantage. SIPE's ability to represent scoping explicitly provides this added flexibility. HACKER [12] and many systems based on logic are flexible in this manner, but SIPE does represent an advance over NOAH and the planners in the NOAH tradition.

3.6 Deductive Operators

In addition to operators describing actions, SIPE allows specification of deductive operators that deduce facts from the current world state. As more complex domains are represented, it becomes increasingly important to deduce the effects of actions from axioms about the world, rather than explicitly representing these effects in operators. Deduction is also necessary for execution monitoring. For example, a sensor might tell the planner that the finger separation of the robot arm has suddenly become zero. The planner must then deduce that the arm is no longer holding block A and that the location of block A is no longer known.

Domain-independent planners that have used a NOAH-like approach (as opposed to a theorem-proving approach) have not had a deductive capability. Although the addition of some deductive capability is straightforward, it is more difficult to find a good balance between expressibility and efficiency. SIPE provides a deductive capability that is useful, but nevertheless keeps deduction under control by severely restricting the deductions that can be made. The importance of this ability will become more significant as execution monitoring capabilities are expanded.

The deductions that are permitted by SIPE have proved to be useful. For example, the PUTON operator in Figure 1 lists only (ON BLOCK1 OBJECT1) as an effect. It does not mention which objects are or are not now CLEARTOP, since that is deduced by deductive operators. Because deductive operators in SIPE may include both existential and universal quantifiers, they provide a rich formalism for deducing (possibly conditional) effects of an action. Effects that are deduced in SIPE are considered to be side effects. (Operators can also specify effects as either main or side effects.) Knowing which ones are side effects is important in handling parallel interactions (see next section).

Figure 2 shows one of the deductive operators in the SIPE blocks world for deducing CLEARTOP relationships. Deductive operators are written in the same formalism as other operators in SIPE, thus permitting the system to control deduction with the same mechanisms it uses to control the application of operators. This also allows constraints to be used and, as this example shows, they play a major role in SIPE's deductive capability.

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 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. The DCLEAR operator in Figure 2 is applied whenever an expansion is produced that has (ON OBJECT1 OBJECT2) as an effect. Deductive operators have no instructions for expanding a node to a greater level of detail. Instead, 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 changing the existing plan. This may "achieve" some goal in the plan (by deducing that it has already been achieved), thereby making it unnecessary to plan actions to achieve it. In Figure 2, matching the precondition will bind BLOCK3 to the block that OBJECT1 was on before it moved to OBJECT2. Since OBJECT4 is constrained to be in the EXISTENTIAL class (see below) and is constrained to not be OBJECT1, the precondition will match (and CLEARTOP of BLOCK3 deduced) only if OBJECT1 is the only object on BLOCK3 (just before moving OBJECT1 to OBJECT2).

The method used for specifying variables as existentially quantified (i.e., constraining them to be in the EXISTENTIAL class) does not provide scoping information. Since only certain types of quantifiers are permitted for efficiency reasons, SIPE interprets preconditions according to defaults that are somewhat non-standard. The scope of each EXISTENTIAL variable appearing as an argument in a predicate is local to that predicate. Each predicate effectively gets a different existential variable. In addition, negated predicates are interpreted as having the quantifier within the scope of the negation. Thus, the variable is effectively universally quantified for negated predicates. As an example, with x declared EXISTENTIAL, the precondition $P(x) \wedge \neg Q(x)$ is interpreted as $\exists x.P(x) \wedge \neg \exists x.Q(x)$ (or equivalently, $\exists x.P(x) \wedge \forall x. \neg Q(x)$). These restrictions make use of SIPE's representation (e.g., the fact that negated predicates are treated differently) to permit handling quantifiers efficiently.

Besides simplifying operators, deductive operators are important in many domains for their ability to represent conditional effects. In NOAH's blocks world, only one block may be on top of another. Consequently, whenever a block is moved, the operator for the move action can be

```
DEDUCTIVE.OPERATOR: DCLEAR
ARGUMENTS: OBJECT1,OBJECT2,BLOCK3 IS NOT OBJECT2,
    OBJECT4 CLASS EXISTENTIAL IS NOT OBJECT1;
TRIGGER: (ON OBJECT1 OBJECT2);
PRECONDITION: (ON OBJECT1 BLOCK3), (NOT (ON OBJECT4 BLOCK3));
EFFECTS: (CLEAR BLOCK3);
```

Figure 2
A Deductive Operator in SIPE

written to state explicitly the effect that the block underneath will be clear. In the more general case in which one large block might have many smaller blocks on top of it, there may or may not be another block on the block underneath, so the effects of the action must be conditional upon this. Since systems like NOAH and NONLIN must mention effects explicitly (universally or existentially quantified variables are not allowed in the description of effects), they cannot represent this more general case with a single move operator. These systems would need two move operators – one for the one-block-on-top case, another for the many-blocks-on-top case. Furthermore, the preconditions for separation of the cases would add an undesirable complication to the representation of the operators.

As the above example shows, SIPE's deductive operators allow existential quantifiers and are powerful enough to handle the general case mentioned. Since SIPE can deduce all the clearing and unclearing effects that occur in the blocks world, the operators themselves do not need to represent them. As domains grow to include many operators, this becomes very convenient. Deductive operators provide a way to distinguish side effects, which can be important. By the use of deduction, more complicated blocks worlds can be represented more elegantly in SIPE than in previous domain-independent planners.

4. Parallel Interactions

As noted before, parallelism is considered beneficial, since optimal plans in many domains require it. (Two segments of a plan are in parallel if the partial ordering of the plan does not specify that one segment must be done before the other.) The approach used in SIPE, therefore, is to keep as much parallelism as possible and then to detect and respond to interactions between parallel branches of a plan. There are three aspects to this situation: recognizing interactions between branches; correcting harmful interactions that keep the plan from accomplishing its overall goal; taking advantage of helpful interactions on parallel branches so as not to produce inefficient plans.

This section first defines helpful and harmful interactions, then describes new features and heuristics in SIPE that aid in handling them. These fall into four areas: (1) reasoning about resources, which is the major contribution of SIPE; (2) using constraints to generate correct parallel plans; (3) explicitly representing the rationale behind each action and goal to help solve harmful interactions correctly; (4) taking advantage of helpful interactions.

SIPE's abilities to handle parallel actions are best described in the context of a sample problem. The canonical simple problem for thinking about parallel interactions is the three-blocks problem. Blocks A, B, and C are on a table or on one another. The goal is to achieve (ON A B) in conjunction with (ON B C), thus constructing a three-block tower. (Initially the two goals are represented as being in parallel.) To move the blocks, there is a PUTON operator (see Figure 1, for example) that puts OBJECT1 on OBJECT2. It specifies the goals of making both OBJECT1 and OBJECT2 clear before performing a primitive move action. (The table is assumed to be always clear and a block is clear only when no block is on top of it.) This problem will be utilized below to provide examples of interactions.

4.1 Defining Helpful and Harmful Interactions

If two branches of a plan are in parallel, an interaction is defined to occur when a goal that is trying to be achieved in one branch (at any level in the hierarchy) is made either true or false by an action in the other branch. Since the actions in a plan explicitly list their effects (a feature shared by many planners, such as NOAH and NONLIN), it is always possible to recognize such interactions. (In a hierarchical planner, however, they may not appear until lower levels of the hierarchy of both branches have been planned.) By requiring that a goal be involved in the interaction, we attempt to eliminate interactions that do not affect the outcome of the plan. For this to succeed, the domain must be encoded so that all important relationships are represented as goals at some level. As we shall see later, this is reasonable in SIPE.

The planner can possibly take advantage of a situation in which a goal in one branch is made true in another branch (a helpful interaction). Suppose we solve the three-blocks problem, starting with A and C on the table and with B on A. In solving the (ON B C) parallel branch, the planner will plan to move B onto C, thus making A clear and C not clear. Now, while an attempt is made to move A onto B in the (ON A B) branch, the goal of making A clear becomes part of the plan. Since A is not clear in the initial state, the planner may decide to make it true by moving B from A to the table (after which it will move A onto B). In this case it would be better to recognize the helpful effect of making A clear, which happens in the parallel branch. Then the planner could decide to do (ON B C) first, after which both A and B are clear and the (ON A B) goal is easily accomplished.

The planner must decide whether or not to add more ordering constraints to the plan to take advantage of such helpful interactions. Ordering the parallel branches sequentially is the best solution to this problem because (ON B C) must be done first in any case, but in other problems an ordering suggested to take advantage of helpful effects may be the wrong thing to do from the standpoint of eventually achieving the overall goal. In general, the planner cannot make such an ordering decision without error unless it completely investigates all the consequences of such a decision. Since this is not always practical or desirable, planning systems use heuristics to make such decisions.

If an interaction is detected that makes a goal false in a parallel branch, there is a problematic (i.e., possibly harmful) interaction, which may mean that the plan is not a valid solution. For example, suppose the planner does not recognize the helpful interaction in our problem and proceeds to plan to put B on the table and A on B in the (ON A B) branch. The plan is no longer a valid solution (*if* it is assumed that one of the two parallel branches will be executed before the other). The planner must recognize this by detecting the problematic interaction. Namely, the goal of having B clear in the (ON B C) branch is made false in the (ON A B) branch when A is put onto B. The planner must then decide how to rectify this situation.

As with helpful interactions, there is no easy way to solve harmful interactions. Here too a correct solution may require that all future consequences of an ordering decision be explored. Stratagems other than ordering may be necessary to solve the problem. For example, a new operator may perhaps need to be applied at a higher level. Consider the problem of switching the values of the two registers in a two-register machine. Applying the register-to-register move operator creates a harmful interaction that no ordering can solve, since a value is destroyed. The solution to this interaction involves applying a register-to-memory move operator at a high level in order to store one of the values temporarily. Correcting many types of harmful interactions efficiently seems very difficult in a domain-independent planner – domain specific heuristics may be required. SIPE simplifies the problem by not shuffling actions between two parallel branches (i.e., the problem must be solvable by leaving things in parallel or placing some number of entire parallel branches sequentially before the others). Although this does prevent some elegant solutions from being found, it retains efficiency while not being overly restrictive.

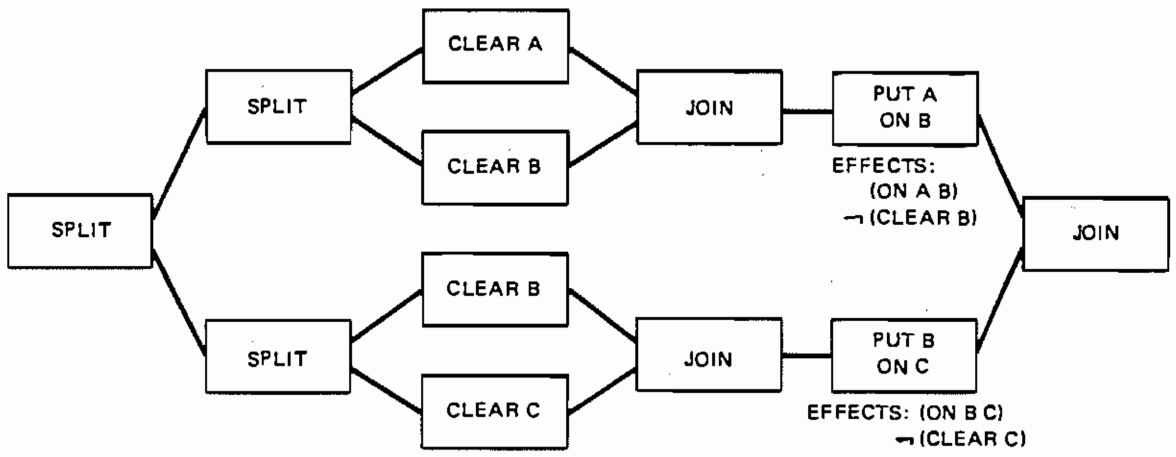

Figure 3
A Plan without Resources

4.2 Reasoning about Resources

The formalism for representing operators in SIPE includes a means of specifying that some of the variables associated with an action or goal actually serve as resources for that action or goal. As we have seen, one advantage of resources is that they help in the axiomatization and representation of domains. Another important advantage of resources is that they facilitate early detection of problematic interactions on parallel branches. The system does not allow one branch to use an object that is also a resource in a parallel branch.

The above example of achieving (ON A B) and (ON B C) as a conjunction shows how SIPE uses resource reasoning to help with parallel interactions. Figure 3 depicts a plan that might be produced by NOAH or NONLIN (or by SIPE without making use of resource reasoning) for this problem. Figure 4 shows a plan from SIPE using resources in the operators.

In NOAH and NONLIN, both original GOAL nodes are expanded with the PUTON operator or its equivalent. This produces a plan similar to the one shown in Figure 3. The central problem is to be aware that B must be put on C before A is put on B (otherwise B will not be clear when it is to be moved onto C). NOAH and NONLIN both build up a table of multiple effects (TOME) that tabulates every predicate instance listed as an effect in the parallel expansions of the two GOAL nodes. Using this table, the programs detect that B is made clear in the expansion of (ON B C), but is made not clear in the (ON A B) expansion. Both programs then solve this problem by doing (ON B C) first.

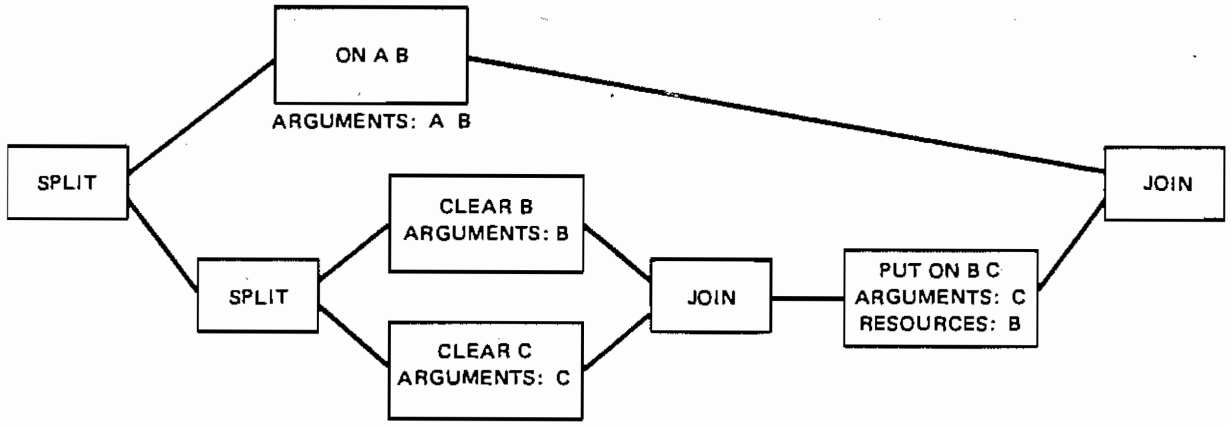

Figure 4
A Plan with Resources

SIPE uses its resource heuristic to detect this problem and propose the solution without having to generate a TOME. (SIPE does do a TOME-like analysis to detect interactions that do not fit into the resource reasoning paradigm.) When some object is listed in an action as a resource, the system then prevents that particular object from being mentioned as either a resource or an argument in any action or goal that is in parallel. In the PUTON operator, the block being moved is listed as a resource in the primitive PUTON action. Thus, as soon as the expansion of (ON B C) with the PUTON operator is accomplished and the plan in Figure 4 produced, SIPE recognizes that the plan is invalid because B is a resource in the expansion of (ON B C) and an argument in (ON A B). This can de detected without expanding the (ON A B) goal at all and without generating a TOME.

This efficient generation of the correct plan depends on listing the block being moved as a resource. This choice is based on domain-dependent heuristic knowledge, and seems reasonable. The robot arm that was moving the block would also be a resource if it were to be represented in the operator. The action of moving an object with an arm causes both the object and the arm to be physically moved; therefore, they are considered to be resources of the action, since nothing in a parallel branch should try to use the arm, move the object, or even be dependent on their respective current locations. The other blocks are not considered resources because they are not used *during* the moving action. Resources can be viewed as a powerful tool that can be utilized by the writer of the operators to represent important domain specific knowledge concerning the behavior of actions.

Not allowing a resource to be mentioned as either a resource or an argument in any action or goal that is in parallel is a strong restriction (though useful in practice). Because this is sometimes too strong a restriction, so SIPE also permits the specification of *shared resources*, whereby a resource in one branch can be an argument in a parallel branch, but not a resource. In the future we would like to have predicates that specify sharing conditions, but this has not yet been implemented.

Resources help in solving harmful interactions, as well as in detecting them. In resource conflicts, no goal is made false on a parallel branch; however, if the resource availability requirements were axiomatized as precondition or goal predicates, an availability goal would be made false on a parallel branch. Thus, resource conflicts are considered to be harmful interactions. SIPE uses a heuristic for solving resource-argument conflicts. Such an interaction occurs when a resource in one parallel branch is used as an argument in another parallel branch (as distinguished from a resource-resource conflict, in which the same object is used as a resource in two parallel branches). This is the type of conflict that occurs in the plan in Figure 4, since B is a resource in the primitive PUTON action and an argument in (ON A B).

SIPE's heuristic for solving a resource-argument conflict is to put the branch using the object as a resource before the parallel branch using the same object as an argument. In this way SIPE decides that (ON B C) must come before (ON A B) in Figure 4. This is done without generating a TOME, without expanding the original (ON A B) node, and without analyzing the interaction. The assumption is that an object used as a resource will have its state or location changed by such use; consequently, the associated action must be done first to ensure that it will be "in place" when later actions occur that employ it as an argument. The above argument may not be convincing, and certainly this heuristic is not guaranteed to be correct, but it is another tool provided by the system that has been proved useful in the four domains encoded in SIPE. By simply setting a flag, the user can prevent the employment of this heuristic if it is inappropriate for a particular domain.

Many interactions that would be harmful in the other systems are dealt with efficiently in SIPE by the resource-reasoning mechanisms. To take full advantage of resources, the system posts constraints. This capability is discussed briefly below. The blocks world example is used above because its simplicity allows the exact details of SIPE's resource reasoning to be explicated. Domains like cooking make more natural and advantageous use of resources to represent problems such as cooking four dishes in three pans on two burners. It should be noted that the concept of resource as implemented is quite limited. For example, we often think of money or computational power as resources, but these could not be represented in our current implementation. We do plan to implement conditions on *shared resources* in the future, which should help express a wider range of resource concepts.

4.3 Constraints and Resource Assignment

SIPE can accumulate various constraints on unbound variables in a plan, which is useful for taking full advantage of resources to avoid harmful interactions. When variables that are not fully instantiated are listed as resources, the system posts constraints on the variables that point to other variables that are potential resource conflicts. When allocating resources, the system then attempts to instantiate variables so that no resource conflicts will occur. (See the OPTIONAL-NOT-SAME constraint in Section 3.) For example, if a robot arm is used as a resource in the block-moving operators, the system will try to use different robot arms (if they are available) on parallel branches, thus avoiding resource conflicts. If only one arm is available, it will be assigned to both parallel branches in the hope that the plan can later be ordered to resolve the conflict. In this way many harmful interactions are averted by intelligent assignment of resources.

4.4 Solving Harmful Interactions

The difficulty entailed in eliminating harmful interactions has already been discussed. However, if the system knows why each part of the plan is present, it can use this information to come up with reasonable solutions to some harmful interactions. Suppose a particular predicate is made false at some node on one parallel branch and true at another node on another parallel branch. Depending on the rationale for including these nodes in the plan, it may be the case that the predicate is not relevant to the plan (an extraneous side effect), or must be kept permanently true (the purpose of the plan), or must be kept only temporarily true (a precondition for later achievement of a purpose). SIPE's ability to specify plan rationale flexibly and to separate side effects from main effects enables it to distinguish these 3 cases accurately, and therefore to represent more problems of this type accurately than could NOAH.

Solutions to a harmful interaction may depend on which of these cases holds. Let us call the three cases side effect, purpose, and precondition, respectively, and analyze the consequent possibilities. If the predicate in conflict on one branch is a precondition, one possible solution is to further order the plan, first doing the segment of the plan that extends from the precondition on through its corresponding purpose. Once this purpose has been accomplished, there will be no problem in negating the precondition later. This solution applies no matter which of the three cases applies to the predicate in the other conflicting branch.

In case both conflicting predicates are side effects, it is immaterial to us if the truth-value of the predicate changes and thus no real conflict exists. In the case of a side effect that conflicts with a purpose, one solution is to order the plan so that the side effect occurs before the purpose; thus, once the purpose has been accomplished it will remain true. When both conflicting predicates are purposes, there is no possible ordering that will achieve both purposes at the end of the plan. The planner must use a different operator at a higher level or plan to reachieve one of the purposes later. However, none of the above suggestions for dealing with interactions can be guaranteed to produce the best (according to some metric, e.g., shortest) solution.

This has been a brief summary of SIPE's algorithm for dealing with harmful interactions. Systems like NOAH and NONLIN do similar things. However, SIPE provides methods for more precise and efficient detection. It should be emphasized that many interactions that would be problematical in the other systems are dealt with in SIPE by the resource-reasoning mechanisms and therefore do not need to be analyzed. When interactions are being analyzed, SIPE requires that one of the conflicting predicates be a goal (not just a side effect) at some level in the hierarchy. In this way, interactions among side effects that pose no problems are not even detected. This requires that all important predicates be recognized as goals at some level, which is easily done in SIPE's hierarchical planning scheme. SIPE provides for exact expression of the purpose of any goal in its operators, which again leads to better analysis of interactions. The system also distinguishes between main and side effects at each node in the plan. This makes it easy to tell which predicates are of interest to us at any level of the plan without looking up the hierarchy (since higher-level goals will become main effects at lower-level actions).

4.5 Achieving Goals Through Linearization

SIPE recognizes helpful interactions and will try to further order the plan to take advantage of them, although the user can control this interactively if he wishes. If a goal that must be made true on one parallel branch is actually made true on another parallel branch, the system will under certain conditions order the plan so that the other branch occurs first (if this causes no other conflicts).

NOAH was not able to take advantage of such helpful effects. (It did have an "eliminate redundant preconditions" critic that eliminated preconditions that occurred twice in the plan, but this could not recognize and react appropriately to a single precondition that was an integral part of the plan being achieved unexpectedly during execution.) NONLIN, on the other hand, did have an ability to order the plan in this way. This is an important ability in many real-world domains, since helpful side effects occur frequently. For example, if parallel actions in a robot world both require the same tool, only one branch need plan to get the tool out of the tool box; the other branch should be able to recognize that the tool is already on the table.

5. Search Control

The automatic search in SIPE is a simple depth-first search to a given depth limit without using knowledge for making choices or backtracking. Obviously, it does not perform particularly well on large problems. The problems involved in planning at the metalevel to control the search are discussed below. The poor performance of automatic search is not debilitating in SIPE, since it has been designed and built to support interactive planning. The user can easily invoke planning operations at any level without being required to make tedious choices that could be performed automatically. In SIPE the user can direct low-level and specific planning operations (e.g., "instantiate PLANE1 to N2636G", "expand NODE 32 with the PUTON operator"), high-level operations that combine these lower-level ones (e.g., "expand the whole plan one more level and correct any problems"), or operations at any level between the two (e.g., "assign resources", "expand NODE 32 with any operator", "find and correct harmful interactions"). This interactive ability is quite useful, since the system can be guided through problems that would not be solved in a reasonable amount of time with the automatic search.

The examples above need not be given to SIPE as text, since the interactive component of SIPE makes use of graphics and has been implemented on a high-resolution black-and-white bitmap display and a color-graphics terminal [7].[2] The planning choices available to the user appear in a menu from which one can be selected by pointing with a mouse or joystick. Similarly, steps in a plan (nodes in the network) can be referred to either by name or by pointing to them.

The procedural networks (plans) produced are presented graphically. The user can choose to view different portions of the plan, do so at different levels of detail, or can look at any alternative plans. The system also provides the information needed by a domain-specific package for displaying the domain configuration. Such a display package has been implemented for the aircraft domain. In this implementation, the user can see a graphic representation of either the actual domain configuration (principally the location of objects) or the one that will exist after some sequence of

[2]The Office of Naval Research Contract N00014-80-C-0300 supported the application of SIPE to the problem of aircraft spotting, during which the implementation of the interactive component was accomplished.

planned steps. The configuration, generally shown together with a plan or partial plan, corresponds to the expected state following execution of that plan.

One feature of SIPE not found in previous domain-independent planners is the ability to explore alternatives in parallel. This is advantageous in an interactive environment because it allows the user to conduct a best-first search easily. In addition to supporting breadth-first and depth-first planning, the interactive planning operations allow *islands* to be constructed in a plan (to arbitrary levels of detail), and then linked together later. The following sections describe the implementation of parallel alternatives and the use of metaplanning as a means of controlling the search.

5.1 Exploring Alternatives in Parallel

A context mechanism has been developed to allow constraints on a variable's value to be established relative to specific plan steps. Constraints on a variable's value, as well as its instantiation (possibly determined during the solution of a general constraint-satisfaction problem), can be retrieved only relative to a particular context. This permits the user to shift focus back and forth easily between alternatives.

SIPE accomplishes this in a hierarchical procedural-network paradigm by introducing CHOICE nodes in the procedural networks at each place an alternative can occur. Constraints are stored relative to choice points. Thus, the constraints on a variable at a given point in a plan can be accessed by specifying the path of choices in the plan that is to be followed to reach that point. Different constraints can be retrieved by specifying a different plan (path of choices). This shifting of focus among alternatives cannot be done in systems that use a backtracking algorithm, in which descriptions built up during expansion of one alternative are removed during the backtracking process before another alternative is investigated. Most other planning systems either do not allow alternatives (e.g., NOAH) or use a backtracking algorithm (e.g., Stefik's MOLGEN, NONLIN). An exception is the system described by Hayes-Roth et al. [4], in which a blackboard model is used to allow the shifting of focus among alternatives.

5.2 Metaplanning

The term *metaplanning* is widely used for referring to reasoning about the planning process itself. The control structure of a planning program (which is continually making choices in an attempt to find a plan) is doing metaplanning, though in SIPE only in an uninteresting way since the control structure is simple and contains little knowledge. (Of course, in SIPE the most abstract operators could be written to make use of metaplanning knowledge in a particular domain.) Many researchers have realized that metaplanning is an important element in being able to control the search efficiently, pointing out that planning about the planning process itself can be done in the same way as planning about the domain, enabling use of a single system architecture for both the planning system and its control structure. As Hayes says in [3]: "We need to be able to describe processing strategies in a language at least as rich as that in which we describe the external domains, and for good engineering, it should be the same language." This section makes two points: (1) metaplanning is a vague concept, and (2) interesting metaplanning appears very difficult in a system that makes the STRIPS assumption.

The idea of metaplanning must be clarified because the term is used in a vague way by many people. A major problem in being more precise about metaplanning is that there is often no clear dividing line between the external domain and the planning process (contrary to Wilensky's argument in [17]). Given any particular system, it will likely be obvious what is at a metalevel in that system. But any particular piece of knowledge might be encoded at either the metalevel or the domain level; furthermore, it is not always clear which is best. It is trivial to convert any domain operator into a metaoperator, and many metaoperators can probably be wired into the domain during design of the domain representation. Wilensky claims that metaplanning can be distinguished by the fact that "meta-goals are declarative structures", but certainly all goals in a planner can be declarative. He claims that another distinguishing characteristic is "meta-goals are domain-independent, encoding only knowledge about planning in general". Metagoals actually contain varying amounts of domain-dependent knowledge. Furthermore, this is necessary, since good planning strategies may differ for differing domains.

For example, consider the advice "use existing objects". This is a fairly domain-independent concept that is used by Sacerdoti in NOAH [9], and mentioned by Wilensky as a metagoal for metaplanning. However, this idea still involves domain knowledge. In the house-building domain, it is desirable to use the same piece of lumber to support both the roof and the sheetrock on the walls. But in another domain, this may not be a good strategy. On the space shuttle, one may want different functions to be performed by different objects so the plan will be more robust and less vulnerable to the failure of any one object. So the "use existing objects" idea makes assumptions about the domain that need to be stated. (Perhaps one wants to apply this idea only to certain portions of the domain.) It would be reasonable to design a system in which such an idea was implemented at the lowest planning level and referred to domain objects (e.g., by automatically adding OPTIONAL-SAME constraints to variables). It would also be reasonable to use this idea in the metaplanner as advice to the "instantiate-variable" planning operator. Thus, the idea contains some domain knowledge and could reasonably be implemented either as part of the domain or at a metalevel.

There are several domain-independent planners in the literature, but none of them does interesting metaplanning. Systems such as SIPE that do hierarchical planning can use abstract operators to encode some metaplanning knowledge, but, as the next paragraph argues, the most interesting metaplanning ideas cannot be encoded in this manner. The only other metaplanning done in such systems is obscure search-control code; no metaknowledge is expressed in the domain-independent formalisms used in these systems for planning in the external domain. There is good reason for this, as these domain-independent formalisms are not adequate for expressing interesting metaknowledge.

These formalisms generally use a *model approach* to representation. (The STRIPS assumption implies a model approach.) Its essential characteristic is that all relationships that hold in the domain are expressed directly (e.g., disjunctions are not allowed), in the sense that the model can be queried in a lookup manner to return an answer quickly about the truth-value of a relationship. This efficient querying ability is, of course, the motivation for the model approach. This approach also means that add and delete lists can be used to solve the frame problem. The disadvantage of the model approach is that many things cannot be represented, since they do not admit to such direct representation. In particular, what we need to say about plans at a metalevel cannot fit easily into the model approach, as it will be hard to represent explicitly (for example) every property of a plan, a failed search branch, an operator, or a constraint that we might want to reason about at the metalevel. The point is that the domain language in these systems is not rich (since it must satisfy this model approach). In particular, it may not be rich enough to do interesting metaplanning, though some metalevel concepts are representable.

6. Execution Monitoring

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. This may involve planning for tests to verify that things are indeed going as expected. Such tests may be expensive (e.g., taking a picture with a computer vision system) so care must be taken in deciding when to use them. The problem of replanning is also critical. 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.

SIPE has addressed only some of the problems of execution monitoring; research is continuing in this area. 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. In accordance with this, the plan can be updated interactively to cope with unanticipated occurrences. Planning and plan execution can be intermixed by producing a plan for part of an activity and then executing some or all of that plan before elaborating the remaining portion.

At any point in the plan, the user can inform the system of a predicate that is now true (though SIPE may have thought it was false). The program will look through the plan and find all the goals that are affected by this new predicate. Since SIPE understands the rationale of nodes in the plan, it can determine how changes affect the plan. For example, if a later purpose is suddenly and unexpectedly accomplished, SIPE can notice the helpful effect and eliminate a whole section of the plan because it knows the preparatory steps are only there to accomplish the purpose. If an unexpected occurrence causes a problem, the system will suggest all the solutions it can find. Problems can be identified because the rationale for goal nodes is given – SIPE can tell which goals must still be maintained and which have already served their purpose. SIPE's repertoire of techniques for finding such solutions is not very sophisticated, however. It includes (1) instantiating a variable differently (e.g., using a different resource if something has gone wrong with the one originally employed in the plan), (2) finding relevant operators to accomplish a goal that is no longer true (and inserting the new subplan correctly into the original plan), and (3) finding a higher level from which to replan if the problems are widespread.

These few techniques are not trivial. Changing an instantiation involves checking the whole plan to see if any parts might be affected by the new instantiation. Inserting a new subplan in the original plan requires a similar check. SIPE's execution-monitoring capabilities are an extension of those in previous domain-independent planners primarily because the explicit representation of plan rationale allows more sophisticated replanning. The deductive capability is also useful for deducing the effects of unexpected occurrences.

Research is continuing in an effort to expand SIPE's execution-monitoring capabilities. After the deductive capabilities of the system have been improved, it will be able to deduce that something is unknown (as will often be the case during execution monitoring). Having unknown quantities will constitute a fundamental modification of SIPE – even the method of determining whether a predicate is true must be changed. SIPE will also be extended to produce conditional plans and to plan for the use of information-gathering actions. Thus, an unknown quantity 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. We also plan to extend the operator description language so that instructions for handling foreseeable errors can be included in operators. Initial investigations show that this ability will include the incorporation of metalevel predicates that are maintained by the system, for subsequent access by the error-handling instructions.

7. Performance of SIPE

SIPE has been tested in four different domains: the blocks world, cooking, aircraft operations, and a simple robotics assembly task. As these domains do not have large branching factors or search spaces, the automatic search can find solutions. The cooking domain was encoded to demonstrate resource reasoning. SIPE operators naturally represented requirements for frying pans and burners during the cooking of a dish. Problems such as cooking four dishes with three pans on two burners were handled efficiently by the resource reasoning mechanisms in SIPE. Handling a problem means producing plans for cooking as many dishes as possible in parallel, with enough serialization to get the task accomplished with the available resources. Such plans consisted of dozens of nodes in our simple cooking world.

The standard blocks world was encoded in SIPE, with some enrichments (e.g., more than one block could be on top of another). Use of deductive operators made the PUTON operator more readable. Resource reasoning enabled SIPE to find and correct parallel interaction problems quickly. All problems of building three- and five-block towers can be solved efficiently as long as they do not involve shuffling of actions between two parallel branches. A number of other problems involving properties of the blocks and quantifiers were also handled elegantly (making use of the constraints in SIPE). For example, the problem of getting *some* red block on top of *some* blue block is easily represented and solved. (SIPE will choose a red block and a blue block that are already clear, if such exist.)

The aircraft operations problem involved planning the launch of a number of airplanes, given their initial locations. This may involve checking for airworthiness, finding paths from a parking space to a fuel station to a runway, and clearing a path if no clear one exists. Problems in this domain included numerous aircraft and long paths, so the plans generated contained hundreds of nodes. Using the automatic search to find paths on the grid would have led to a combinatorial explosion, so path variables were instantiated to a list of locations by a special-purpose generator. SIPE's ability to allow operators to loop over a list enabled the system to generate easily the goals of clearing the locations specified in the list produced by the generator. Using these abilities, SIPE was able to generate large plans automatically for correctly launching many planes from parallel runways.

The robotics problem encoded was similar to a blocks-world problem except that the utilization of the robot arm was planned and the mating of two parts represented. (This means the system must deduce that, when an object moves, all things mated with it move conjointly.) SIPE was adequate for representing this problem.

8. Conclusion

SIPE's operator description language was designed to be easy to understand (to enable graceful interaction) while being more powerful than those found in previous domain-independent planners. Constraints, resources, and deductive operators all contribute to the power of the representation. Deductive operators allow quantified variables and can therefore be used to make fairly sophisticated deductions, thus eliminating the need to express effects in operators when they can be deduced. They are also useful in distinguishing main effects from side effects.

One of the most important features of SIPE is its ability to constrain the possible values of variables. It is well known that this enables more efficient planning, since choices can be delayed until information has been accumulated. Other advantages of constraints, however, are also critical. A key consideration is that constraints allow convenient expression of a much wider range of problems. Constraint satisfaction finds variable instantiations efficiently by taking advantage of the fact that invariant properties of objects are encoded in the sort hierarchy. Constraints also help prevent harmful parallel interactions.

SIPE incorporates several new mechanisms that make it easier to deal with the parallel-interaction problem. The most significant of these mechanisms is the ability to reason about resources. It has been beneficial in user interaction to have reasoning about resources as a central part of the system, because resources seem to be a natural and intuitive way to think about objects in many domains. Combined with the system's ability to post constraints, resource reasoning helps the system to avoid many harmful interactions, to recognize sooner those interactions that do occur, and to solve some of these interactions more quickly. SIPE'S handling of interactions is also improved by its ability to differentiate side effects and to determine correctly the rationale behind actions.

SIPE provides more flexibility than its predecessors in specifying the rationale behind a plan. It is able to specify explicitly the scope of a condition and the node that accomplishes higher-level goals. This is useful for determining how long a condition must be maintained and what changes in the world cause problems in the plan, for finding solutions to such problems, and for determining the relationship among different levels in the hierarchy. SIPE also provides new mechanisms for interfacing between two different levels of hierarchical descriptions.

A major difference between SIPE and previous planners is that SIPE is interactive. Its interactive capabilities help the user guide and direct the planning process, thereby allowing alternative plans to be explored concurrently by means of the context mechanism. Thus, the user can shift focus as he pleases without being required to understand the program's search strategy or backtracking algorithm.

SIPE is not yet robust enough to operate in complex real-world situations. Its execution-monitoring capabilities are currently being expanded. Future research needs to investigate iterative plans, conditional plans, and uncertainty, since these cannot currently be represented.

ACKNOWLEDGMENTS

Many people influenced the ideas expressed in this paper. Special thanks go to Ann Robinson who helped design and implement SIPE, and to Nils Nilsson, Stan Rosenschein, and Michael Georgeff for many enlightening discussions.

REFERENCES

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

2. Grosz, B. et al., "TEAM: A Transportable Natural-Language System", Technical Note 263, SRI International Artificial Intelligence Center, Menlo Park, California, 1982.

3. Hayes, P.J., "In Defense of Logic", *Proceedings IJCAI–77*, Cambridge, Massachusetts, 1977, pp. 559-565.

4. Hayes-Roth, B., Hayes-Roth, F., Rosenschein, S., and Cammarata, S., "Modeling Planning as an Incremental, Opportunistic Process", *Proceedings IJCAI–79*, Tokyo, Japan, 1979, pp. 375–383.

5. McDermott, D., "A Temporal Logic for Reasoning About Processes and Plans", Cognitive Science, forthcoming.

6. Robinson, A.E., and Wilkins D.E., "Representing Knowledge in an Interactive Planner" *Proceedings of the First Annual Conference of the AAAI,* Stanford, California, 1980, pp. 148–150.

7. Robinson, A.E., Final Report on the SPOT Project, SRI International Artificial Intelligence Center, Menlo Park, California, forthcoming.

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

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

10. Sridharan, N., and Bresina, J., "Plan Formation in Large, Realistic Domains", *Proceedings CSCSI Conference,* Saskatoon, Saskatchewan, 1982, pp. 12-18.

11. Stefik, M., "Planning with Constraints", *Artificial Intelligence 16(2),* 1981, pp.111-140.

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

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

14. van Melle, W., "A Domain-Independent Production-Rule System for Consultation Programs", *Proceedings IJCAI–79,* Tokyo, Japan, 1979, pp. 923–925.

15. Vere, S., "Planning in Time: Windows and Durations for Activities and Goals", Jet Propulsion Lab, Pasadena, California, November 1981.

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

17. Wilensky, R., "Meta-planning", *Proceedings AAAI–80,* Stanford, California, 1980, pp. 334-336.